AD-772 809

ARCHITECTURAL PRINCIPLES FOR VIRTUAL COMPUTER SYSTEMS

Robert P. Goldberg

Harvard University

Prepared for:

Electronic Systems Division

February 1973

Unclassified
Security Classification

AD 772 809

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Harvard University Cambridge, MA 02138	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED
	2b. GROUP N/A

3. REPORT TITLE
ARCHITECTURAL PRINCIPLES FOR VIRTUAL COMPUTER SYSTEMS

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
None

5. AUTHOR(S) (First name, middle initial, last name)
Robert P. Goldberg

6. REPORT DATE February 1973	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
8a. CONTRACT OR GRANT NO F19628-70-C-0217	9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-73-105	
b. PROJECT NO.		
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
d.		

10. DISTRIBUTION STATEMENT
Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES Thesis Div. Engineering & Applied Physics Harvard University	12. SPONSORING MILITARY ACTIVITY Deputy for Command and Management Systems Hq Electronic Systems Division (AFSC) L G Hanscom Field, Bedford, MA 01730

13. ABSTRACT

The thesis develops principles for designing machines to support virtual computer systems. The virtual computer system (VCS) is an important construct which arises as a solution to a problem of present computer system technology.

The most important new result of the thesis is the model of a process running on a virtual computer system (VCS) and the derivation of design principles from that model. The approach adopted is to consider the introduction of VCS's into the rich, complex architectures likely to be found in IV generation systems. Because of the additional system structure in the IV generation, the somewhat ad hoc third generation virtual machine software mapping techniques should be doomed to failure. This result leads us away from an interpretation of virtual machines that depends implicitly on techniques used in third generation systems. Instead, we are able to develop a generalized model of a process running on a IV generation VCS. The model allows us to understand different properties of virtual machines and to interpret a number of proposed implementations of VCS's in terms of the model. Furthermore, the model leads naturally to an implementation of virtual machines, the Hardware Virtualizer (HV) which provides an efficient and simplified mechanism for virtual machines. A number of detailed examples illustrate how the Hardware Vitualizer might operate in an actual IV generation system.

I 72

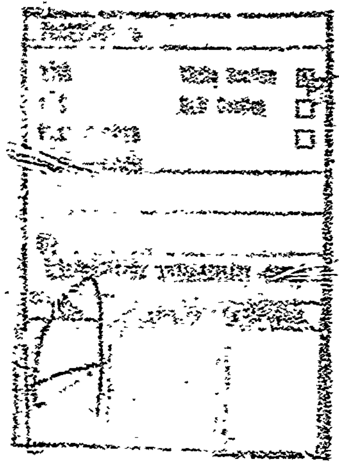
DD FORM 1 NOV 65 1473

Unclassified
Security Classification

248

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
virtual computer system						
virtual machine						
computer systems architecture						
operating system design						
virtual memory						
segmentation						
paging						

/C


[illegible]

ESD-TR-73-105

ARCHITECTURAL PRINCIPLES FOR VIRTUAL COMPUTER SYSTEMS

Robert P. Goldberg

February 1973

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

(Prepared under Contract No. F19628-70-C-0217 by Harvard University
Cambridge, Massachusetts 02138.)

FOREWORD

This report was prepared in support of Project 2801 by Harvard University, Cambridge, Massachusetts under Contract F19628-70-C-0217, monitored by Major Roger R. Schell, HQ. ESD(MCIT), and was submitted January 16, 1973.

This technical report has been reviewed and is approved.

[illegible]

MELVIN B. EMMONS, Colonel, USAF
Director, Information Systems Technology
Deputy for Command & Management Systems

ABSTRACT

The thesis develops principles for designing machines to support virtual computer systems. The virtual computer system (VCS) is an important construct which arises as a solution to a problem of present computer system technology. The most important new result of the thesis is the model of a process running on a virtual computer system and the derivation of design principles from that model. The approach adopted is to consider the introduction of VCS's into the rich, complex architectures likely to be found in IV generation systems. Because of the additional system structure in the IV generation, the somewhat ad hoc third generation virtual machine software mapping techniques should be doomed to failure. This result leads us away from an interpretation of virtual machines that depends implicitly on techniques used in third generation systems. Instead, we are able to develop a generalized model of a process running on a IV generation VCS. The model allows us to understand different properties of virtual machines and to interpret a number of proposed implementations of VCS's in terms of the model. Furthermore, the model leads naturally to an implementation of virtual machines, the Hardware Virtualizer (HV), which provides an efficient and simplified mechanism for virtual machines. A number of detailed examples illustrate how the Hardware Virtualizer might operate in an actual IV generation system.

PREFACE

I am deeply indebted to my advisor and friend, Dr. U. O. Gagliardi for his wisdom and kindness. It was Dr. Gagliardi's encouragement and confidence in me at several crucial points during my academic career that allowed the research to continue to a successful conclusion. In particular, collaboration with Dr. Gagliardi on an earlier paper, "Virtualizeable Architectures" led to some key insights that have become an important part of the thesis.

I would also like to express my gratitude to the other members of my committee, Professors T. E. Cheatham, Jr. and R.V. Book, and Mr. G. H. Mealy for the time they have invested in reading the thesis and for their pertinent comments. In addition, Mr. Mealy's question (several years ago) of whether a virtual 360/67 could be run under CP-67 essentially started the research on this thesis.

I would like to thank my colleagues at both MIT and Harvard for the numerous discussions about virtual machines over the years. Particular note for their stamina is due to J.P. Buzen, H.S. Schwenk, S.E. Madnick, A.L. Brown, and R.M. Kierr.

I would also like to thank A.W. Armenti, S.W. Galley, and J.F. Haverty for a very careful reading of the thesis. Any felicities of style can be traced to their persistence and suggestions.

Three people aided in the preparation of the thesis and

sincere thanks is due them. They are Lloyd Dickman who suggested and executed some of the figures in Chapter 3, my cousin Paul Goldberg who drew the figures in Chapter 2 and calculated and plotted the data in the graphs of Chapter 4, and my wife Judith Goldberg who prepared the remainder of the figures.

Because of the long duration of the work, portions of the research were carried out at MIT Lincoln Laboratory and MIT Project MAC as well as Harvard University Center for Research in Computing Technology. It is a pleasure to cite this support and acknowledge the encouragement given by J.F. Nolan, J.J. Fitzgerald, and J.C R. Licklider.

I would like to express my gratitude to my family and friends, especially my mother and sister, for their sincere interest and good wishes. Finally, my wife, Judy, deserves special recognition for her remarkable patience and support during the preparation of the thesis. Her many significant contributions, both real and virtual, are greatly appreciated.

[This research was supported in part by AF19(628)-5167, DAHC15-69-C-0347, and F19628-70-C-0217. The thesis was prepared using the Honeywell MULTICS System.]

TABLE OF CONTENTS

LIST OF FIGURES

SYNOPSIS

The thesis develops principles for designing machines to support virtual computer systems. The virtual computer system (VCS) is an important construct which arises as a solution to a particularly vexing problem of present computer system technology.

While the recent development of large multi-access, multi-programming, multi-processing systems has simplified and improved access to computer systems by the bulk of the user community, there has been one important class of users unable to profit from these recent advances. This class is the system programmers whose programs must be run on a <u>bare</u> machine and not on an <u>extended</u> machine, e.g. under the operating system. System programs, e.g. other operating systems or different versions of the same operating system, require direct addressability to the resources of the system and do not call upon the operating system to manage these resources. Thus, system programs may not co-exist with normal production uses of the system. This situation has forced most installations into clumsy and inefficient administrative solutions such as "stand-alone" debugging.

Recently, a technique for resolving these difficulties has been devised. The solution utilizes a construct called a virtual computer system or virtual machine which is, basically, a very efficient simulated copy (or copies) of the bare host machine. These copies differ from each other and from the host only in

their exact configurations, e.g. the amount of memory or particular I/O devices attached. Therefore, not only standard user programs but system programs and complete operating systems that run on the real computer system will run on the VCS with identical results (except for certain special timing dependencies). Thus, the VCS provides a generalization over familiar systems by also being a multi-environment system.

The recent origin of the field has forced the development of a terminology to represent ideas in VCS's. Chapter 2 introduces some basic notions in VCS's and proposes terminology to represent them. The terminology is new but some excerpts were previously published by the author in "Virtual Machines: Semantics and Examples" [53].

Despite the rather significant benefits that derive from virtual machines, only a limited number of current systems feature this facility. This is largely due to the unsuitable nature of existing hardware. In order to support a VCS on contemporary equipment, a particular software construction must be employed. However, this construction may only be applied to machines with certain properties. Chapter 3 examines the construction of VCS's on third generation computer systems and derives a set of empirical requirements to determine if a machine may be virtualized. The appendices treat the application of these empirical rules to a number of case studies. This material is new although some preliminary results were reported by the author in "Virtual Machine Systems" [54] and "Hardware

Requirements for Virtual Machine Systems" [52].

The remainder of the thesis (Chapter 4) develops a structure for computer systems that permits the orderly introduction of VCS's into future computer systems. IV generation computer systems will likely feature a formal implementation (in firmware) of the process model. Because of the existence of a large database needed by the firmware (and privileged software) to support the process model, it will be difficult to employ the software mapping techniques used in III generation VCS's. A model is developed to represent the execution of processes on a IV generation VCS. The model features two maps: (1) a process map called ϕ which maps process names, e.g. segments, semaphores, process-id's, into resource names, e.g. memory locations, processor numbers, and (2) a virtual machine map (VMmap) f which maps virtual resource names into real resource names. The process map is strictly an intra-level map expressing a relationship within a virtual machine; the VMmap is an inter-level map expressing a relationship between (the resources of) two adjacent levels of (virtual) machines. Thus, the action of running a process on a VCS consists of running it under the composed map $f \circ \phi$. If the VCS itself is running a VCS under it, then the action consists of running a process under $f \circ f \circ \phi$. Thus, for VCS recursion, only the f map must be invoked recursively, not the process map ϕ. This result implies that for a complex ϕ but a rudimentary f, recursion should be easy.

The IV generation VCS model leads directly to a design, the Hardware Virtualizer (HV), for proposed implementations of IV generation VCS's. The design has the characteristic that all process exceptions are handled directly within the executing VCS without software intervention. All resource faults (VM-faults) by a VCS are directed to its Virtual Machine Monitor (VMM) without knowledge of processes on the VCS. Fault handling, invocation and execution of VCS's works directly regardless of recursion.

Detailed illustrations are provided for several possible choices of the VMmap f, e.g. relocation and bounds (R-B), paging. Preliminary performance estimates indicate that for an R-B VMmap f (with associative memory), VCS performance will be approximately that of the real machine, regardless of the depth of recursion. A paged VMmap f provides performance comparable to the real machine for several levels of recursion over a range of likely operating conditions.

Although the model and proposed implementations of Chapter 4 arise in an attempt to guarantee virtualization for IV generation architectures, by suitable simplification and interpretation of the model, the principles which emerge are applicable to other architectures, as well. In particular, the model suggests introducing paging in the IBM 360/67 as a formal f-map, rather than an ad hoc ϕ-map as was done. This leads to a greatly simplified structure for CP-67.

All of the material presented in Chapter 4, the IV generation VCS model, suggested implementations, performance expectations, examples, etc., is new and original. An early proposal for a firmware-assisted implementation of VCS's was described by the author [with U.O. Gagliardi] in "Virtualizable Architectures" [51]. However, the interpretation of that proposal in Chapter 4, its strengths and weaknesses in terms of the IV generation VCS model, is new and original.

CHAPTER 1. INTRODUCTION

1.0 OVERVIEW

The Need for Virtual Machines

Recent developments in computer system design have been directed toward improving the ease of computer use by the normal user population. Some of the more dramatic developments include the introduction of large multi-access, multi-programming, and multi-processing systems. These systems have eliminated the need for physical access to the main computing facility by the users, and for direct software access to the system resources by the users' programs. Thus, users sit at teletypes or submit jobs via remote card readers and need not be concerned either with the status of lights or switches on the central processor, or with the allocation of memory, I/O devices, or processor time to their programs. These functions are now performed (transparently to the users) by the operating system. When either users or their programs require services to be performed, they call upon the operating system which performs the actual manipulation and allocation of resources. In some sense, the operating system (together with the hardware) provides an idealized or _extended_ view of a computer sytem. Thus, program preparation, debugging, and running has become significantly easier for most of the user population.

Unfortunately, one very important group of users has been unable to reap the benefits of these developments and advances. These are the users who, for a variety of reasons, desire or need direct access to the resources of the system. Their programs are written to run on a bare machine. Calling upon an operating system (the so-called extended machine) will not do for them. We will term such users system programmers and the programs they desire to run, system programs. Because system programs cannot run under the normal operating system in the production environment, it is usually necessary to treat them as special cases. When they must be run, the normal operating system is brought down and a system program may run "stand-alone." This practice is inconvenient for both the system programmers and the normal users, and inefficient and wasteful of the system resources. Furthermore, it can introduce additional personnel and procedural complexities. Indeed, it is ironic that those who develop and maintain operating systems are often the very people who take the least advantage of their efforts.

A few specific examples of some system programming tasks might clarify these points.

(1) Debugging the operating system-- The operating system is written to run on the bare machine. Currently, debugging and improving the operating system is treated as a second class activity, often assigned to computer time in the middle of the night.

(2) Running diagnostic software-- The software for

testing malfunctions in the various units in the system, e.g., is a tape drive recording correctly, is a disk drive seeking correctly, etc. normally runs stand-alone on the bare machine or on a rudimentary monitor (Test and Diagnostic Monitor) which itself runs on a bare machine.

(3) Running other operating systems-- When there is more than one operating system for a given computer system (or even different releases of the same operating system) it is often easier to run a subsystem (compiler, interpreter, etc.) under its own operating system than to take the time and cost to convert it to another operating system.

What are Virtual Machines?

Recently, a technique for resolving these difficulties has been devised. The solution utilizes a construct called a virtual computer system (VCS) or virtual machine. [Section 2.1 covers most of this terminology.] Basically a virtual machine is a very efficient simulated copy (or copies) of the bare host machine. Because of its functionality, it is possible to run system programs; because of its efficiency, it is reasonable to run them in the normal production environment. The program which mediates between the virtual machine and the actual resources of the system (the real or host machine) is called the virtual machine monitor (VMM). Since the virtual machine is identical to the host machine in all significant respects, the VMM need not employ the familiar instruction-by-instruction software interpretation technique that normally must be used. Rather, virtual machines may be constructed in such a way that most instructions of the virtual machine execute on the host directly. Just those instructions which cannot be permitted to execute directly must be interpreted. Thus, we incorporate these notions into our definition of a virtual computer system in order to distinguish it from a number of other objects which have often been casually called virtual machines.

> A virtual computer system is a hardware-software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute on the host processor in native mode.

Thus, a VCS provides an efficient operation of one or more copies of a complete computer system, similar to the host (or real) system. These copies differ from each other and from the host only in their exact configurations, e.g. the amount of memory or the particular I/O devices attached. The virtual and real processors must either be identical or members of the same computer family, e.g. IBM System/360-370. Therefore, not only standard user programs but system programs and complete operating systems that run on the real computer system will run on the VCS with identical results [except for certain special timing dependencies]. Thus the VCS provides a generalization over the familiar multi-access, multi-programming, multi-processing systems, by also providing a multi-environment system.

In addition to permitting the three system programming examples, above, to coexist with normal production uses of the system, virtual machines introduce an additional degree of system flexibility. This flexibility is a consequence of the additional level of binding in a virtual machine, between the resources referenced by system programs running on the VCS and the actual resources that they correspond with. Some immediate advantages are:

(1) Running with a virtual configuration which is different from the real configuration-- This use can be important for running systems with more virtual memory or processors than actually exist, or debugging computer networks and telecommunications applications.

(2) Measuring operating systems-- Since the VMM mediates between the virtual and real resources of a system, it can measure how an operating system on a virtual machine is manipulating its resources (without requiring a modification to the operating system).

(3) Adding hardware enhancements to a system-- If hardware enhancements, e.g. paging, are added to the host machine, it may still be possible to run a virtual machine which does not incorporate these enhancements. Under these conditions the operating system running on the virtual machine need not be modified. Thus, with relatively simple VMM modifications, the hardware enhancements can be utilized to provide improved resource handling and capabilities.

Current Virtual Machine Understanding

Despite the rather significant benefits that derive from virtual machines, only a limited number of current systems feature this facility. This situation is a result of (1) the recent origin of the concepts, (2) misunderstandings of the techniques for implementing virtual machines, and (3) the largely unsuitable nature of existing hardware. Thus, since existing computer systems were not designed to support virtual machines, trying to implement them is largely a hit-or-miss proposition.

In the thesis, we explore the problems associated with implementing virtual machines on contemporary hardware. By developing an empirical basis and a set of hardware rules for determining which existing systems support virtual machines, we are able to verify the inadequacy of current designs [Chapter 3].

At last, there is a growing realization of the importance of designing machines which are virtualizable, i.e. support virtual machines. To date, there seem to be two opposing schools of thought. While both views have some validity, both have significant disadvantages.

The first view [55], typified by Lauer and Snow [73], argues that since the virtual machine must have all the functionality of a real machine, the simpler the real machine is the easier will be the virtual machine construction. [See Section 2.4 and 4.8.] In particular, they suggest eliminating supervisor state, memory mapping, etc. What they introduce instead is a special relocation-bounds type memory map, in which the absolute contents of the register may be added to but not read or written. Such an approach greatly simplifies virtual machine construction, but it has as a major weakness that the virtual machine's programming environment is rather severe and devoid of structure. This fact implies that the development of large, interesting systems on the virtual machine may be difficult. The authors observe that their proposal "... lacks some of the important features of modern systems, particularly segmented virtual memories and a suitable parameter passing mechanism..." [76].

The second view, introduced by U.O. Gagliardi and the author [51] argues that in complex (likely IV generation) computer systems with a firmware implementation of the process model and layered segmented address structure, there will be a need for firmware support of virtual machines as well. [See Section 2.4 and 4.4.] What emerges is a proposal which directly supports the image of a process executing on a virtual machine. The major advantage of this proposal is that it is applicable to very complex computing environments that are likely to be developed in the near future. The principal disadvantage is that, since this approach does not propose a direct hardware resource map, a certain amount of software intervention is still required. In addition, there are some fundamental limitations on the kinds of recursive structures (running a VMM on a virtual machine) that can be supported.

Results of the Research

The principal result of the thesis is the development of a model for running a process on a complex (IV generation) computer system [Section 4.2]. The model identifies a formal resource map (virtual machine map) and distinguishes it from the more familiar process map. This allows us to understand a number of very complex phenomena in a relatively simple way, and permits us to interpret the two earlier proposals [51,76] as merely special (sub-optimal) applications of the model [Sections 4.4 and 4.8]. Furthermore, the model leads us directly to an optimal proposed implementation [Section 4.5] which is, in some sense, a synthesis of the better ideas of the two previous proposals. Thus, the proposal is applicable to the complete range of computer systems, including the complex (IV generation) future systems, yet retains the directness and simplicity of a hardware resource map which needs little software support. In addition, since this implementation is derived directly from the model, the recursive invocation of virtual machines, i.e., running a VMM under a VMM etc., poses no additional problems. Furthermore, the implementation should be very efficient. While performance measures are hard to compare, there is evidence that for certain likely choices of maps, the performance of the virtual machine will be extremely close to that of the real machine. Although the model and proposed implementations of Chapter 4 arise in an attempt to guarantee virtualization for IV generation achitectures, by suitable simplification and interpretation of the model the principles which emerge are applicable to other generations of architectures [Section 4.8].

Perhaps the major contribution of the thesis is a clear understanding of the architectural principles for virtual machine support. By following the guidelines set forth in the thesis, the designer of a future computer system can be assured that his machine will be virtualizeable. Thus, the dissemination of these principles should have a highly significant impact upon both the theoretical and practical aspects of computer science.

> Computer architecture, like other architecture, is the art of determining the needs of the user of a structure and then designing to meet those needs as effectively as possible within economic and technological constraints. Architecture must include engineering considerations, so that the design will be economical and feasible: but the emphasis in architecture is upon the the needs of the user, whereas in engineering the emphasis is on the needs of the fabricator [21].

1.1 PLAN OF THE THESIS

The remainder of the thesis is divided into four chapters. Each chapter is preceded by an introduction which outlines the material to follow, but perhaps some brief organizational remarks here will be helpful.

The most important new result of the thesis, the model of a process running on a virtual computer system (VCS) and the derivation of design principles from that model, is presented in Chapter 4. This chapter is largely self-contained, and the reader knowledgeable about current virtual machine applications and technology should be able to skip directly to Chapter 4, if desired. The approach adopted in this chapter is to consider the introduction of VCS's into the rich, complex architectures likely to be found in IV generation systems. Because of the additional system structure in the IV generation, the somewhat ad hoc third generation virtual machine software mapping techniques should be doomed to failure [Sections 4.1 and 4.3]. This result leads us away from an interpretation of virtual machines that depends implicitly on techniques used in third generation systems. Instead, we are able to develop a generalized model of a process running on a IV generation VCS [Section 4.2]. The model allows us to understand different properties of virtual machines and to interpret a number of proposed implementations of VCS's in terms of the model [Section 4.4]. Furthermore, the model leads naturally to an implementation of virtual machines, the Hardware Virtualizer (HV) which provides an efficient and simplified mechanism for virtual machines [Section 4.5]. A number of detailed examples illustrate how the Hardware Virtualizer might operate in an actual IV generation system [Section 4.6]. In addition, the principles developed in this chapter are generally applicable to other architectures as well as the highly complex IV generation. Proper simplification and interpretation of the model leads to retrospective principles for how the second and third generation systems <u>should</u> have been constructed to support virtual machines [Section 4.8].

The earlier chapters develop a basic understanding of virtual machines and existing techniques for implementing them, and provide additional motivation for seeking the breakthroughs described in Chapter 4. In Chapter 2 we introduce some of the basic virtual machine terminology and properties, and survey existing implementations and related literature. The approach we take to terminology in Chapter 2 is largely descriptive and qualitative. Later, in Section 4.7, a number of these notions are re-cast in terms of the VCS model of Chapter 4.

Chapter 3 examines the existing technology for third generation virtual machines and shows why and how it is inadequate. From the construction used in IBM's CP-67 [Appendix A] we derive the general software implementation of third generation VCS's. A close scrutiny of typical third generation systems leads to a series of empirical hardware requirements for virtualizable third generation architectures [Section 3.2]. Application of these rules in a number of case studies indicates that most third generation systems cannot be virtualized

[Appendix B]. This result points out the general unsuitability of third generation architectures (for virtualization) and the weaknesses of the software construction technique. Thus, we are forced to look elsewhere for the key issues and solutions in designing virtualizable architectures [Chapter 4].

Other material presented in Chapter 3 includes requirements for hybrid virtual machines (HVM) [Section 3.3], ad hoc improvements to third generation hardware [Section 3.4], software requirements for Type II (extended machine host) virtual machines [Section 3.5], and suggested virtual machine software primitives [Section 3.6]. Additional third generation subjects are provided in the first three appendices which deal with a CP-67 tutorial [Appendix A], case studies of some third generation machines [Appendix B], and case studies of some third generation Type II operating systems [Appendix C].

Chapter 3 should be of particular interest to two classes of readers. The first class includes those individuals running third generation systems and contemplating the introduction of a virtual machine facility. The various empirical rules and studies should help to determine if virtual machines are possible on the reader's system and what practical options may be taken. The second class of readers includes computer system designers who can observe how relatively insignificant design decisions have rendered most third generation machines unvirtualizable. It will be necessary for these designers to avoid such errors in designing the virtual machine support for the IV generation.

Chapter 5 gives a very brief summary of the research and points out several promising areas for additional study.

The final material of the thesis, Appendix D, is a glossary of some of the terms used in the thesis. In addition to the specialized terminology for virtual machines, e.g. virtual machine recursion property, a number of general though not widely-known terms, e.g. interior decor, are also defined.

CHAPTER 2.

VIRTUAL COMPUTER SYSTEMS

Terminology and Background

2.1 PLAN OF CHAPTER 2

Chapter 2 is divided into four sections. The first section introduces some of the basic terminology that is used in the thesis. Since virtual computer systems have only recently come into existence, we have been forced to develop new terms to represent ideas about VCS's. The approach taken in this section is somewhat descriptive. Later, in Section 4.2, a number of these notions are recast in terms of the VCS model of Chapter 4.

In the second section, we contrast the notion of a virtual machine with a number of related concepts. Next, examples of existing virtual computer systems are cited. In the final section, published literature relating to virtual machines is briefly surveyed.

2.1 TERMINOLOGY FOR VCS'S

The term Virtual Computer System has been loosely applied to a range of different computer system organizations. Recently, as noted in Chapter 1, its use has been reserved for a specific type of computer system entity.

> A Virtual Computer System is a hardware-software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute directly on the host processor in native mode.

There are two parts to this definition.

Environment-- A virtual computer system must simulate a real existing computer system. Programs and operating systems which run on the real system must run on the virtual system with identical effect. Since the simulated machine may run at a different speed from the real one, timing dependent processor and I/O code may not perform exactly as intended. This is very similar to the limitation, however, that characterizes interchangeability of programs among different members of a "compatible" computer family, such as between the IBM 360/40 and the 360/75.

Implementation-- Most instructions being executed must be processed directly by the host CPU without recourse to instruction by instruction interpretation. This guarantees that the virtual machine will run on the host with relative efficiency. It also compels the virtual machine to be similar or identical to the host, and forbids tampering with the control store to add an entirely new order code.

The VCS and/or host configurations are made up of one or more central processors, main memory, and I/O devices (and possibly I/O processors too). Usually the virtual computer system configuration will have only one central processor. That processor may be called a _virtual machine_. However, the literature often informally uses virtual machine and virtual computer system, VM and VCS, as interchangeable. We shall do likewise.

Since a VCS is a hardware-software duplicate of a "real existing computer system" there is always the notion of a _real computer system_, RCS, or _real machine_, RM, whose execution is functionally equivalent to the VCS.

The program executing on the host machine that creates the VCS environment is called the _Virtual machine monitor_, _VMM_. CP-67 [9,65,85], which is discussed below, gives the VMM the generic name of "control program". Only in the discussion of CP-67 will we depart from the use of VMM.

Self-virtualizing vs. Family-virtualizing VMM

The VCS differs from the host only in its exact configuration, e.g. the amount of memory available or the particular I/O devices attached. A further distinction might be in the precise characteristics of the virtual processor. The "implementation requirement" in the VCS definition, above, implies that the virtual processor must be similar or identical to the host processor. If they are not identical, then the virtual machine must be a member of the same processor family as the host. For example, the virtual and host processors might be different models of a compatible product line, e.g. 370/155 and 360/30, 360/67 and 360/65, or Data General Supernova and Nova. The virtual and host processors might even be the same model processor, but only differ in the fullness of the instruction set implemented, e.g. universal instruction set vs. standard instruction set.

This distinction may be captured by the following two categories:

Self-virtualizing (SV)-- The virtual machine is identical to the host.

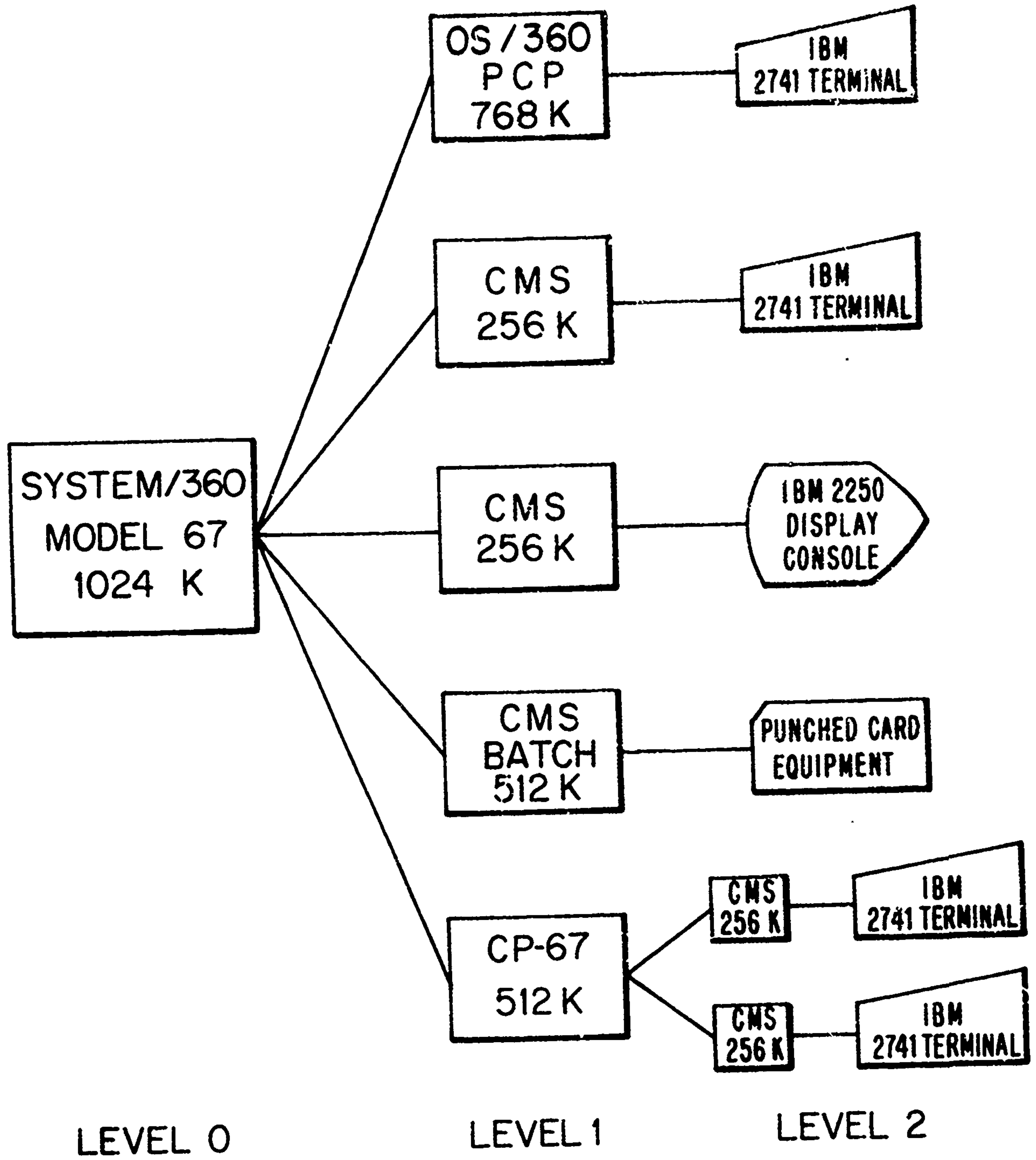
Family-virtualizing (FV)-- The virtual machine is a member of the same computer family as the host.

In a self-virtualizing VCS, the functionally equivalent real computer system is identical to the host. Therefore, we sometimes call the host machine, the real machine. We then say that the real/host machine has a virtualizable architecture [51].

If a VCS is self-virtualizing, then it is possible to run another copy of the VMM on the VCS, thus, producing another level of virtual machines. This demonstrates the virtual machine recursion property and is of distinct practical importance since it allows alterations to be made to the VMM running on the virtual machine and permits testing the VMM in the normal operating environment. [See Section 4.2 for the development of VM recursion in terms of the VCS model.] We say that the VMM defines the level of recursion or virtualization. If a VMM is running on a level n VM, then it produces a level n+1 VM. In particular, the real/host machine is level 0. In Figure 2-1, the 512K virtual 360/67 running CP-67 is level 1 and its two virtual machines are level 2.

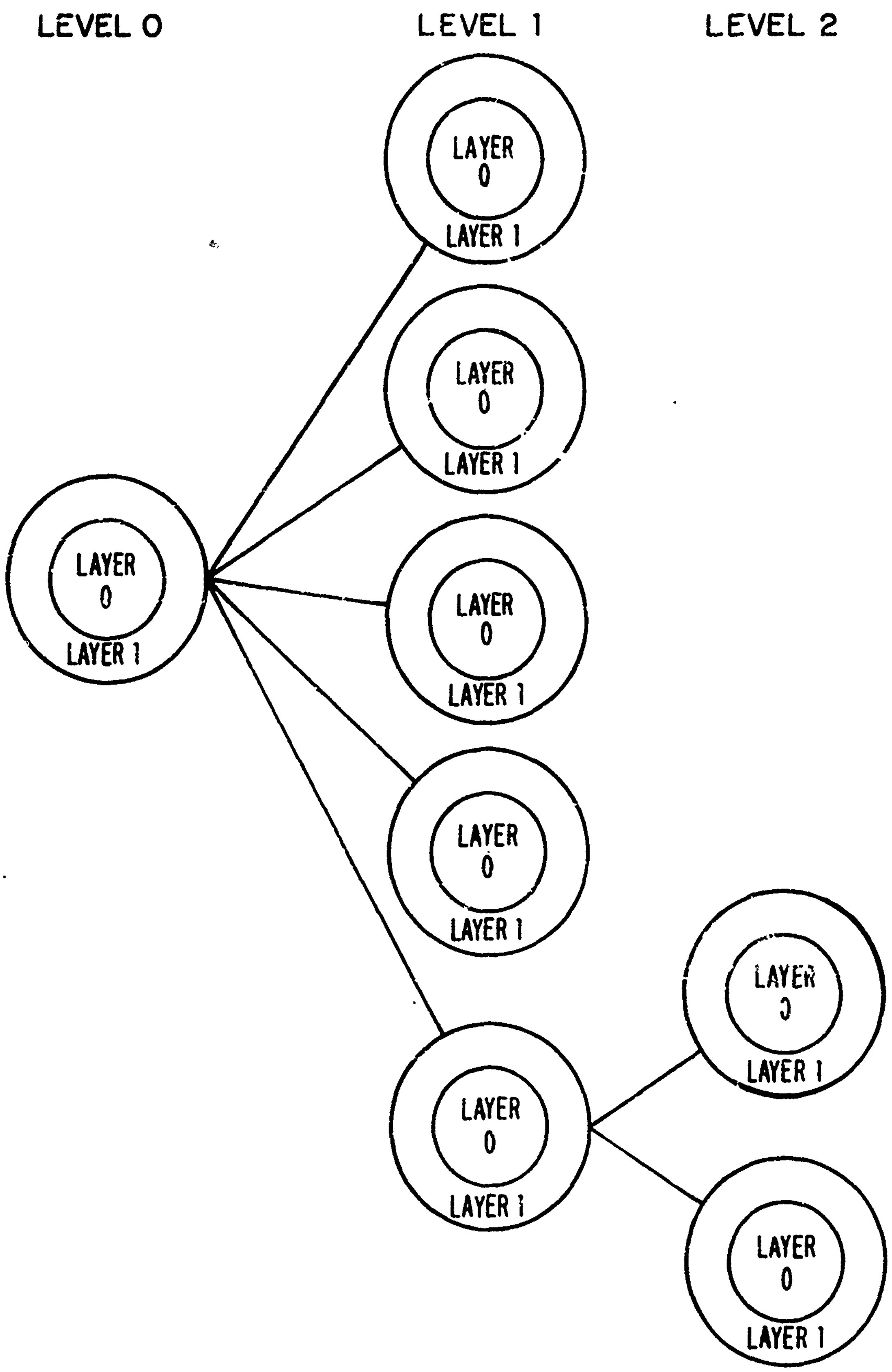
Level vs. Layer

The notion of VCS levels should not be confused with the concept of layers in a computer system. As will be discussed in Chapters 3 and 4, III and IV generation computer systems have structures which implement layers of restricted access to data or instructions. In III generation systems, the two layers are usually called Master/Slave mode (supervisor/problem, Executive/User, etc.). In IV generation systems, the layers will likely be called rings [57,106]. Since a layer is a relationship between two parts of an individual VCS and a level is a relationship between two VCS's, they are somewhat independent notions. In particular, each level of a VCS must appear to have the same number of layers as the real machine. However, as will be seen later, [Section 4.3] software implementation of VM's have utilized the restricted access of layers to simulate the effect of levels. This is merely an implementation technique and not a general principle. These issues are discussed further in "Virtualizeable Architectures" [51] and Section 4.2. Figure 2-2 illustrates the relationship between levels and layers for the CP-67 VCS shown in Figure 2-1.



TYPICAL VCS — CP-67

FIGURE 2-1



LEVEL vs. LAYER IN III GENERATION VCS

FIGURE 2-2

Virtual Machines and Related Constructs

The implementation requirement of the VCS definition indicates that a "statistically dominant subset of the virtual processor's instructions execute directly on the host processor in native mode." To provide functional equivalence between the VCS and real computer system RCS, those instructions which do not execute directly must be simulated on an instruction-by-instruction basis. Thus, the "performance" of the virtual machine is bound by the real machine and by a "complete software interpreter machine," CSIM. Rather than set any precise performance objective or require any myopic machine dependent implementation as part of the definition of virtual machine, we prefer to treat VCS as a broad class of potential systems. [However, for a particular system under consideration, we normally apply the term VCS if the direct execution subset is maximal.]

Thus, the "complete software interpreter machine" is not a VCS since no instruction of the CSIM executes directly. Neither is the real machine. Although all instructions of the RM execute directly, the RM is not a hardware-software duplicate, and indeed it requires no VMM.

A recently distinguished VCS entity is the hybrid virtual machine HVM. The HVM is a VM in which all supervisor state, e.g. Master Mode or Ring 0, instructions are interpreted. The HVM has been found to be a useful and easy-to-construct artifact where more familiar VCS techniques have failed. [See Section 3.3 and Appendix B.]

An example of the way each of these four constructs might execute a particular instruction sequence on a III generation computer system is indicated in Figure 2-3. Successive instruction executions, i.e. time, are represented from left to right and the two states shown for each of the four constructs are software interpretation and direct execution. Thus, the real machine is always shown in the direct execution state while the virtual machine occasionally traps to software interpretation for certain instructions.

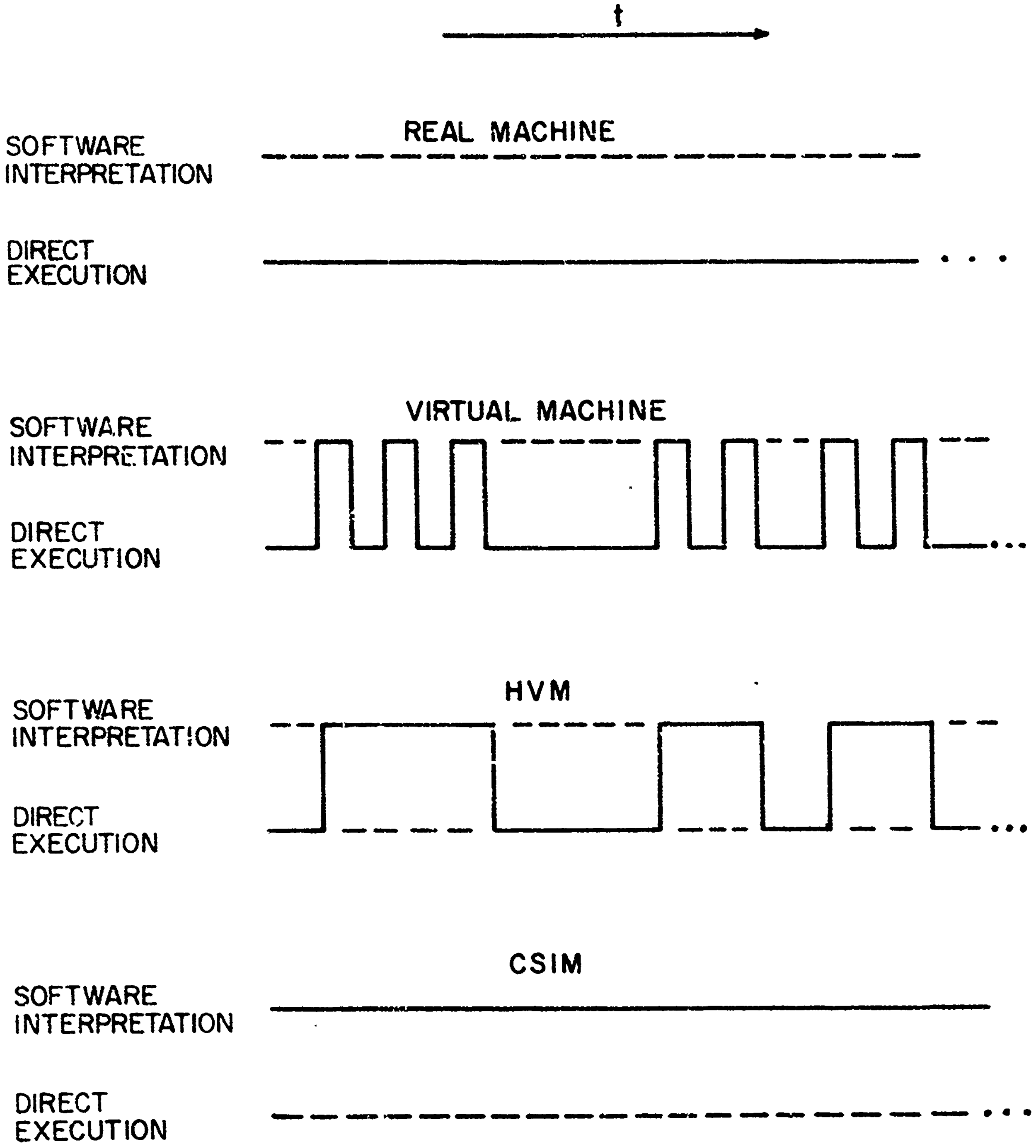
Type I vs. Type II VMM

The implementation requirement specifies that most VCS instructions execute directly on the host. It does not indicate how the VMM gains control for that subset of instructions which must be interpreted. This may be done either by a program running on the bare host machine or by a program running under some operating system on the host machine. In the case of running under an operating system, the host operating system primitives may be used to simplify writing the virtual machine monitor. Thus, two additional VCS categories arise:

Type I--The VMM runs on a bare machine.
Type II--The VMM runs on an extended host [53,75], under the host operating system.

In either case, the virtual machine being created is equivalent to the bare host or a related family member.

Type I (bare host) and Type II (extended host) VCS's are

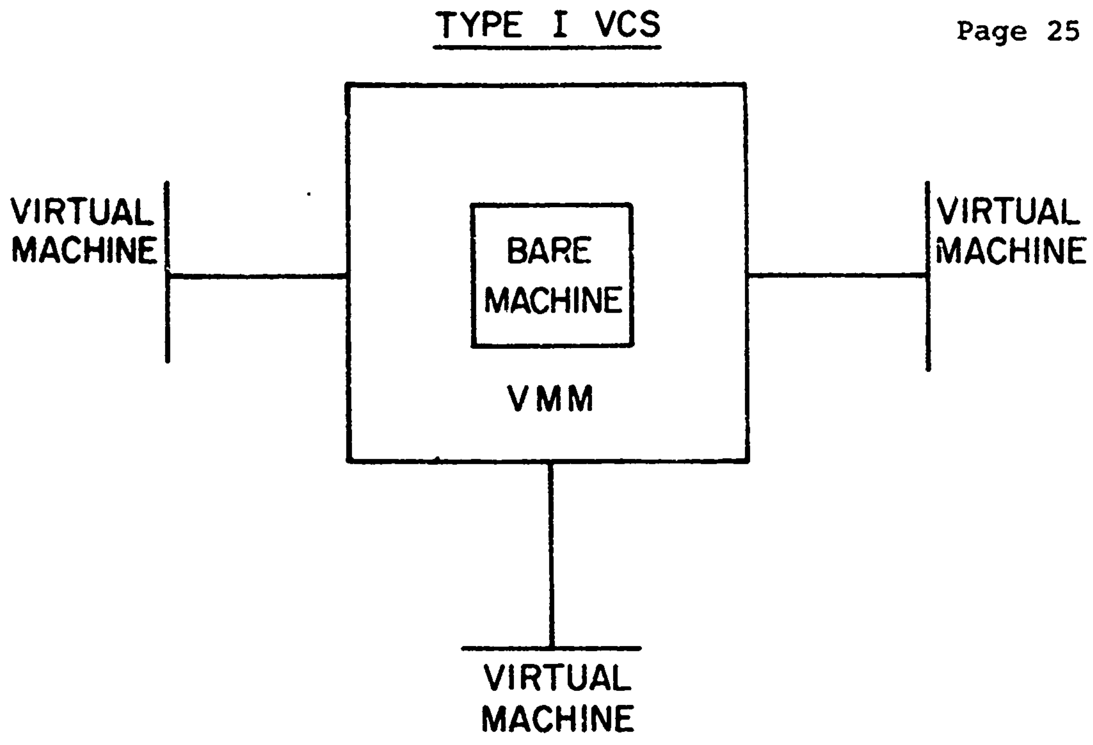


VIRTUAL MACHINE vs. OTHER CONSTRUCTS
FIGURE 2-3

illustrated in Figure 2-4. In both Type I and Type II VCS, the VMM creates the VCS environment. However, in a Type I VCS, the VMM on a bare machine must perform the system's scheduling and (real) resource allocation. Thus, the Type I VMM may include much code not specifically needed for a VCS. In a Type II VCS, the resource allocation and environment creation functions for VM's are more clearly split. The operating system does the normal system resource allocation and provides a standard extended machine environment. The VMM program runs on the extended machine environment and produces a pseudo-hardware environment. [Another interpretation of Type I and Type II virtual machines is presented in Section 4.2.]

There are different advantages to be derived from using either a Type I or Type II VCS. If an installation normally runs one particular operating system and most users would prefer to treat the system as an extended machine, then a Type II VMM may be used to provide a virtual machine for the occasional user who may desire it. He might want to either debug system code or, say, run some foreign operating system. In such a case, the access to the system is likely to be more convenient for the numerous typical users, and, probably, more efficient since there is less overhead in running the preferred operating system. [Another approach to achieving some of these objectives is reported by Parmelee [93].]

If there are numerous alien operating systems to be run and there is no such thing as a preferred environment, it may make sense to run a Type I VCS. Of course, the decision to choose



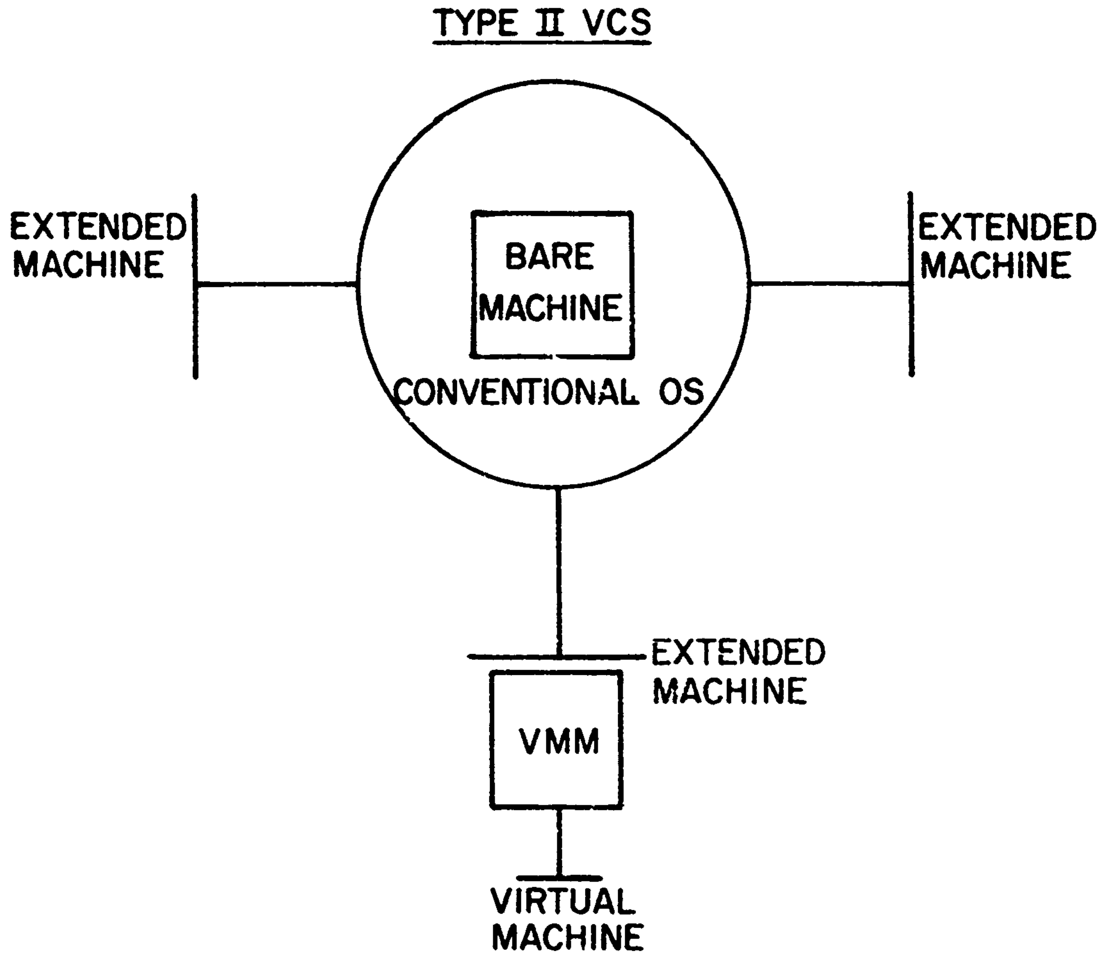


FIGURE 2-4 TYPE I vs TYPE II VCS

Type II over Type I depends also on the existence of an operating system that is capable of supporting a Type II system. Another important factor to consider is the amount of work required to bring up either system.

Type I and Type II VCS's are two quadrants of a more general tabular structure which deserves future study. See Figure 2-5. The two rows of the table indicate the host environment on which the monitor program runs. The two columns indicate the target environment produced by the monitor. A Type I VMM runs on a hardware host to produce a hardware target, i.e. virtual machine. A Type II VMM runs on an extended host to produce a hardware target. The boxes marked EMM stand for Extended Machine Monitor. The Type I EMM is a conventional operating system. The Type II EMM runs under one operating system to produce the interface generated by some other operating system. Type II EMM can be a valuable tool for transporting software when a VCS is not used [36,45].

		TARGET	
		BARE	EXTENDED
HOST	BARE	TYPE I VMM	TYPE I EMM
	EXTENDED	TYPE II VMM	TYPE II EMM

MACHINE ENVIRONMENTS

FIGURE 2-5

2.2 COMPARISON WITH RELATED NOTIONS

Because of the recent development of the field of computer systems architecture, and the absence of a generally used, as well as agreed upon, terminology, it is possible that the notion of a virtual machine may be confused with certain other related ideas. In this section, some common misconceptions regarding virtual machines will be cited to provide some additional insight into the key notions involved in a virtual machine.

2.2.1 Virtual Machine vs. Virtual Memory

Virtual memory refers to an addressing system in which a logical address generated by a program, called a virtual memory address, is mapped into some other, possibly different, physical memory address. Virtual memory may be implemented in numerous ways, among them simple relocation, multiple relocation, paging, segmentation, or a combination of these methods [38]. If paging or segmentation is employed, it is often the case that the amount of virtual memory exceeds actual physical memory.

Since one aspect of a real computer system is memory, the corresponding attribute of a virtual computer system is customarily called virtual memory [51]. The virtual memory need not be larger than available physical memory, although it may be, if implemented on a machine with paging. This is the case with CP-67 [9,65,85]. The virtual memory may be even mapped identically into all or part of its correspondingly addressed physical memory. This is the case with the proposed CP-65

[54,58], experimental IBM 360/30 [71], or Japanese work [50]. The permitted virtual memory size or method of relocation (or not) are issues which affect the flexibility of the use of virtual memory, or ease of performing system storage allocation. These issues are largely tangential to the notions involved in virtual machines.

One particular connection between virtual machines and virtual memory concerns the interrupt control register locations that most systems have fixed in low physical core. Since the memory of a virtual machine is functionally equivalent to the memory of a real machine, virtual interrupt locations in virtual memory must have the same apparent effect for the virtual machine as the corresponding real locations have for the real machine.

2.2.2 Virtual Machine vs. Virtual Machine Time-Sharing System

A Virtual Machine Time-Sharing System (VMTSS) is a timesharing system in which each user's interface with the system is a virtual machine [81]. It is possible to have a virtual machine without having timesharing or a VMTSS. The experimental IBM 360/30 [71] or the Japanese HITAC 8400 [50] are examples of virtual machines running on a non-timesharing system. The virtual 360 under UMMPS [60] is an example of a Type II virtual machine running under a (conventional) timesharing system that is not a VMTSS. Possible confusion between virtual machines and VMTSS's may be related to the success of CP-67 which is a VMTSS.

2.2.3 Virtual Machine vs. Pseudo Machine

A pseudo-machine [99], also called an extended machine [53] or a user machine [75], is a composite machine produced through a combination of hardware and software, in which the machine's apparent architecture has been changed slightly to make the machine more convenient to use. Typically, these architectural changes have taken the form of removing I/O channels and devices, and adding system calls to perform I/O and other operations. These system functions are normally invoked by transferring to some location in virtual memory, as in Multics [86,92], or issuing a Supervisor Call (SVC) or Master Mode Entry (MME), etc. instruction with suitable address code. The supervisor interprets the SVC, executes the desired function, and returns to the user.

In a virtual machine, the machine's apparent architecture is identical to a real machine's architecture. The visibility of all I/O channels and devices is left intact. There are no supervisory functions provided. If a machine wishes to perform I/O, it must utilize its own I/O programs. Indeed, the system does not provide an SVC handler for the virtual machine.

Possible confusion between virtual machine and pseudo machine can be attributed to an overuse of the term "virtual machine" by certain authors. Too often "virtual machine" has been used to denote a non-hardware "user" machine [111,112].

2.2.4 Virtual Machine vs. Emulated Machine

Emulation is a technique that has been used successfully to map one system architecture into another different architecture [63,82,110]. Emulation at the interior decor interface (hardware interface) causes the layer zero software visibility of the native system to (almost) include visibility of the target system. Similarly, emulation at the exterior decor interface (extended machine) extends the software visibility of, say, layer 1 to include software visibility of layer 1 of the target system [51].

Well-known examples of emulators include IBM's 1401 on the 360/30, 7094 on the 360/65, and DOS under OS on the 370/145. In the first two examples, the target and native interior decors are vastly different. This requires the emulator to be implemented via special additional microcode, and the processor must operate in this non-native mode.

In a virtual machine, on the other hand, the target and host are either similar or identical hardware machines. If a self-virtualizing machine has an emulator implemented on it, then it should be accessible to the virtual machine in the same way that it is to the real one. Thus, if there exists a special instruction, such as Do Interpretive Loop (DIL) [110] which puts the machine into an emulation mode, then its execution by a virtual machine should put the virtual machine into emulation mode. [IBM has just announced this facility for VM/370 [67].]

2.3 EXAMPLES OF VIRTUAL MACHINES

There are relatively few examples of virtual machines that may be cited. We will show later [Section 3.2] that this is due to the general unsuitability of contemporary designs and hence the significant requirements placed on a computer system that is to support a software construction of virtual machines.

2.3.1 IBM Research M44/44X-- Type I FV (Family-virtualizing)

The IBM M44 [91,103] was a highly modified (II generation) IBM 7044, extended to include paging and a number of operational modes, such as Problem/Supervisor, Location Test/No Location Test, and Mapping/No Mapping. Through the use of an operating system, called MOS (Modular Operating System), it produced a set of virtual machines called 44X's. [These 44X's are not virtual machines in the strictest sense since they are slightly different from real 7044's.] A 44X operating system runs on each 44X and provides the customary services found on most operating systems. The M44/44X system was very important, in the history of computing, for trying out a number of innovative ideas, among them, virtual machines.

2.3.2 IBM Cambridge Scientific Center CP-40-- Type I FV

CP-40 [3,27,59], the forerunner to CP-67, was constructed on an IBM 360/40 that was specially modified through the addition of an associative memory paging box. This system was a VMTSS and supported fourteen virtual 360's. These 360's were standard

360's and did not contain any of the special modifications of the host model 40. The virtual 360's were able to run numerous 360 operating systems, including the standard OS/360 [66,84] and CMS [85]. a conversational system which was developed at Cambridge Scientific Center specifically for use with CP-40. The system suffered from the composite limitations of the 360/40's processor speed, a modest amount of real core, and a slow disk for paging. However, CP-40 did serve to demonstrate the viability of the virtual machine notion. Furthermore, it provided a VMTSS, and with CMS, a convenient conversational system for the 360 at a time when no other general purpose timesharing system for that series of machines was working. Indeed, success with this project led directly to the larger system, CP-67.

2.3.3 IBM Cambridge Scientific Center CP-67-- Type I FV

CP-67 [9,65,85] was begun as an experimental system in conjunction with MIT Lincoln Laboratory in January, 1967. Initially, the CP-40 system was modified slightly to support the different memory relocation system found on the 360/67. Subsequently, the entire CP-67 system was rewritten. CP-67 is a VMTSS which runs on the 360/67 and supports between thirty and forty users. It is, by far, the most widely known virtual machine system. The virtual 360's produced under CP-67 have been very successful in running a number of different 360 operating systems. These include (but are not limited to) OS/360 (in many versions), DOS, DOS-APL, CMS, LLMPS [85]. The virtual machines have been used in telecommunications applications. The

University of Grenoble has even run the IBM OS/ASP (Attached Support Processor) system, connecting a real 360/40 to a virtual 360 [9].

The principal limitations of the virtual machines produced by CP-67 are that a small set of programs may not work correctly. These are:

(1) Programs which depend on precise execution times;

(2) Programs which depend on the relationship between processing time and input/output time,

(3) Programs which depend on precise real-time, and

(4) Programs which contain self-modifying channel programs.

The first two restrictions are identical with those to insure compatibility between two different member processors of the 360 family [69].

2.3.4 Extension of CP-67 to support virtual 360/67-- Type I SV

In May, 1969, the author designed a slight extension to CP-67 to support a relatively efficient implementation of a virtual 360/67 (with virtual relocation) [54]. With slight modifications, this design of a virtual 360/67 was implemented in the fall of 1969 by F. Furtek. Independently of this work, A. Auroux at the IBM Cambridge Scientific Center also designed a virtual 360/67 extension to CP-67 [95]. The two designs are very similar and the Auroux implementation has become part of the standard (IBM distributed) CP-67 system.

The virtual 360/67 has been limited to a 24-bit addressing mode single processor machine. With this capability, various 360/67 operating systems have been run, including CP-67 and TSS/360 [95]. That is, CP-67 has been run under itself. In fact, this chain has been tested to a depth of at least four. The principal additional limitation on the operation of the virtual 360/67 over a standard virtual 360 (under CP-67) is that programs that dynamically alter the contents of segment or page tables may not execute as intended. The results may or may not be identical with a real 360/67. In a real machine, the contents of the associative registers may affect the result and hence results may also be unpredictable.

2.3.5 Virtual 360 under UMMPS-- Type II FV

The virtual 360 under UMMPS [4,60] is a Type II virtual machine system. UMMPS is a conventional multiprogramming system which was written at the University of Michigan to run on a dual processor 360/67. The University of British Columbia has made a few modifications to UMMPS to support a virtual 360. These modifications have taken the form of some additional supervisor call facilities to perform some crucial operations in dispatching the virtual machine. Apparent exceptional conditions occurring in the virtual machine are reflected by the UMMPS supervisor back to a user program which performs, for example, the simulation of privileged instructions. Virtual (standard) 360's are the only virtual machines supported by this system, although other user interfaces besides the virtual machine interface are provided (such as MTS or a conventional batch system). Extensions to

UMMPS to support virtual 360/67's as well as the standard 360's are being designed.

2.3.6 Japanese Electrotechnical Lab. HITAC-8400-- Type II FV

The HITAC-8400 is the Japanese manufactured version of the RCA Spectra 70/45. It is a conventional third generation computer without a memory mapping system, similar to the IBM System/360. As an aid in debugging ETSS [49], a new time-sharing system for the HITAC-8400, a Type II virtual machine was constructed on the existing, vendor-supplied operating system, TOS/TDOS [50]. The virtual machine system is very similar to the virtual 360 constructed on UMMPS. One interesting aspect of this effort was the attempt at simulating various virtual I/O devices that did not have exact physical counterparts (such as a display scope). This was done, of course, since the virtual machine was being used entirely to debug supervisory code for ETSS and not for running production jobs under it.

2.3.7 Hardware Modified IBM 360/30-- Type I FV

A virtual machine system has been constructed on an experimental 360/30 through a combination of software and hardware, i.e. microprogramming modifications [71]. These modifications, which have been introduced in order to simplify virtual machine construction, are used to implement a special "monitor" mode, and to relocate the interrupt control area from the regular control program area in physical page 0. The system is not a VMTSS and only supports one virtual machine. Its principle use is in monitoring and evaluating the performance of existing operating systems such as Operating System/360 (OS/360), and the Basic, Disk, and Tape Operating Systems (BOS, DOS, TOS/360).

2.3.8 MIT Project MAC PDP-10 ITS-- Type II SV HVM

As will be seen in Chapter 3, it is not possible to implement virtual machines on the DEC PDP-10 using conventional third generation software techniques. Recently, the author and S. W. Galley have introduced the notion of a hybrid virtual machine for the PDP-10 [56]. With this technique, all executive-mode instructions are interpreted while all user-mode instructions execute directly. The MIT Project MAC Dynamic Modeling-Computer Graphics PDP-10 has provided an ideal environment for this implementation. The ITS (Incompatible Timesharing System) provides the requisite features to support a Type II (extended machine host) VCS. Furthermore, the hardware configuration is blessed with sufficient core and, at times, enough CPU cycles to make it possible to run the HVM. [See discussion in Appendix C.]

2.3.9 Grenoble Monitor System for Standard 360/40

GMS, Grenoble Monitor System, [58] was designed by the IBM Grenoble Scientific Center to provide multiple 360's on a standard IBM 360, i.e. one without relocation. The major objective of the system was the concurrent support of a limited number of virtual machines, each running CMS, the Cambridge

Monitor System. Since a standard 360 does not provide a memory relocation system, the store and fetch memory protection system must be used in virtual machine construction. [See Appendix B.] However, because of difficulties in installing fetch-protect on a European model 40, the actual implementation had to compromise the pure 360 hardware definition of the virtual machines. This led to a system in which modifications were made to CMS to allow it to run in this "enhanced" environment.

2.3.10 IBM VM/370

In July, 1972, IBM announced a virtual machine facility for various models of the System/370 [67]. This system seems to be a direct descendant of CP-67. When delivered, VM/370 will be the first formally-supported, commercially-offerred VCS available from any manufacturer.

2.4 RELATED LITERATURE

The early literature on virtual machines dealt almost exclusively with the implementation mechanisms, policies, or applications of one particular virtual computer system. The treatment of policies, e.g., scheduling algorithms, page replacement algorithms, in these papers involves largely the same considerations as in non-virtual machine timesharing or multiprogramming systems. Therefore, they will not be reviewed here. Many of these papers, particularly those devoted to memory management appear in the annotated bibliography of Parmelee et al [95]. Since the thesis is concerned with developing architectural principles and mechanisms, we will not review the virtual machine applications papers either. However, some of the more significant papers are listed in the bibliography.

The virtual machine implementation and mechanism papers go back to an early unpublished paper by Sayre [101,103] which describes the IBM M44/44X work and its implications. Perhaps the first virtual machine report to get wider circulation was the description of CP-40 produced by the IBM Cambridge Scientific Center in 1966 [3].

With the arrival of CP-67, a number of papers were published which describe the mechanisms used to create virtual machines. These papers include Field [47] and Auroux and Hans (in French) [9] which are the most complete. Later papers by IBM authors, e.g. Meyer and Seawright [85], repeat much of the same published material.

Two early papers deserve particular note. Fuchi's paper [50] reports on the implementation of the first Type II VMM and also gives insightful suggestions for hardware "improvements" to facilitate virtual machine construction. Keefe's paper [71] reports on some hardware modifications attempted with an IBM 360/30 in order to simplify production of a non-self-virtualizing VMM.

Among the author's early papers are the first papers that discussed self-virtualizing systems and the development of an empirical basis for deciding which machines are self-virtualizing. "Virtual Machine Systems" [48] reports on the design of a virtual 360/67 under CP-67 and a virtual 360/65 under "CP-65" (which runs on a standard 360/65). This same report and "Hardware Requirements for Virtual Machine Systems" [52] contain the first published analyses of the problems involved in implementing a VMM on a specific class, i.e. third generation machines, rather than an individual computer system.

Recently, there has been increased interest in VCS"s. At the (September) 1971 IEEE International Computer Society Conference, the author organized a session of papers [53,94,113] and discussion devoted solely to VCS's. One of those papers, written by the author, "Virtual Machines: Semantics and Examples" [53] proposed some of the terminology introduced in Chapter 2.

At the 1972 ACM International Computing Symposium at Venice, three papers specifically related to VCS's appeared. In "Virtual Input/Output in a Virtual Environment" [8], Ancilotti, Cavina and Liltmaer of Pisa investigate the possibility of allowing an I/O channel to execute a program allocated in virtual memory. "Is Supervisor-state Necessary?" [76] by Lauer and Snow of the University of Newcastle-upon-Tyne is a particularly insightful and well-written paper proposing the organization of a self-virtualizing computer system based upon a relocation and bounds form of address relocation and no supervisor state. As will be discussed in Section 4.8, this design is a special sub-case of the hardware virtualizer of Section 4.5 with $f=R-B$ and ϕ=identity map.

The third paper was "Virtualizeable Architectures" [51] by U.O. Gagliardi and the author. This paper was the first paper to suggest the need for firmware support for virtual machines in a complex computing environment. The design that emerged, called the Venice Proposal (VP) in this thesis, features a hardware Virtual Machine Identifier (VMID) Register to specify the active virtual machine. It was the work with U.O. Gagliardi on the VP that led the author directly to the more general model of a VCS presented in Section 4.2. In Section 4.4, the VP is presented and interpreted as a special case of the VCS model.

A number of other papers, not specifically related to VCS's, have affected the author's thinking during the course of this research. Some of these papers discuss hierarchical operating systems, layered protection structures, or deferred resource binding. The early paper by Dennis and Van Horn [1,40] describes an hierarchical operating system and suggests primitives for interprocess control. The paper by Dijkstra [41] describes a view of a system seen as a layered structure. The paper by

Schroeder and Saltzer [106] proposes a hardware mechanism sufficient to support a layered system structure. Evans and LeClerc [46,77] describe a hardware addressing structure in which addresses, e.g. segment numbers, are assigned relative to each individual procedure call. Finally, Schell's doctoral thesis [104] discusses the relationship between logical and physical resources and the mechanisms for electrical binding.

CHAPTER 3.

PRINCIPLES FOR III GENERATION VIRTUAL COMPUTER SYSTEMS

Inadequacy of Current Designs

3.0 PLAN OF CHAPTER 3

Chapter 3 develops principles for third generation virtual computer systems. The approach adopted is somewhat empirical and relies on a scrutiny of existing III generation hardware and software systems. The chapter is divided into four subject areas: (1) Characteristics of III generation hardware, (2) Hardware requirements for third generation virtual computer systems, (3) Software requirements for Type II third generation VCS's, and (4) Conclusion.

In the first section, we summarize relevant architectural characteristics of III generation systems.

Section 3.2 generalizes the virtual machine construction used in CP-67 to derive a set of empirical hardware requirements for determining if a third generation architecture is virtualizable. [Appendix A provides a brief tutorial on CP-67.] The rules are applied to a number of representative III generation machines and the results are summarized in a table. [Appendix B provides the detailed analysis of these case studies.] In Section 3.3 we introduce the hybrid virtual machine (HVM) and develop its set of less stringent hardware requirements. Finally in Section 3.4, modest ad hoc "improvements" are considered to make more third generation machines virtualizable.

Section 3.5 develops a series of empirical operating system requirements for determining if an operating system on a third generation machine will support a Type II (extended machine host) VMM. The rules are applied to a number of representative III generation operating systems and the results are summarized in a table. [Appendix C provides the detailed analysis of these case studies.] In Section 3.6, we propose an example of a set of idealized operating system primitives to support Type II virtual machines.

Section 3.7 reviews some of the results of the chapter and provides some perspective of how this work relates to Chapter 4.

3.1 CHARACTERISTICS OF III GENERATION COMPUTER SYSTEMS

One of the most significant architectural innovations of III generation computer systems is the existence of two processor modes of operation. These two modes, variously called supervisor and problem, executive and user, or master and slave provide two degrees of privilege for programs. Thus, operating system programs, when executing in master mode, may typically invoke certain special instructions, while user programs, operating in slave mode, may not. These instructions normally control such critical system features as the input/output facilities and the setting of privilege itself. Attempts by programs in slave mode to issue privileged instructions are normally prevented in one of several ways.

Another architectural characteristic of III generation systems is, often, the presence of a supervisory call instruction. Since a user program is prohibited from issuing privileged instructions, the operating system normally provides a means for users to call upon it for these services. Entry to the operating system is usually via a special instruction, such as Supervisor Call (SVC, BRM, UUO, MME) which traps to a particular location in the monitor. The monitor checks the request, performs (if permitted) the desired function and returns to the user program.

An additional characteristic of the III generation computer system concerns the means of addressing memory. Usually, but not always, some kind of relocation and/or protection system is provided. The memory relocation may take the form of simple relocation and bounds (R-B), paging, or even segmentation and paging. Most of these systems provide an absolute addressing mode when the memory relocation system is disabled. This usually occurs when master mode is entered.

III generation I/O systems usually communicate with the CPU via asynchronous interrupts which are affected by an interrupt control area in low physical core. Interval timers are set and notify the CPU similarly.

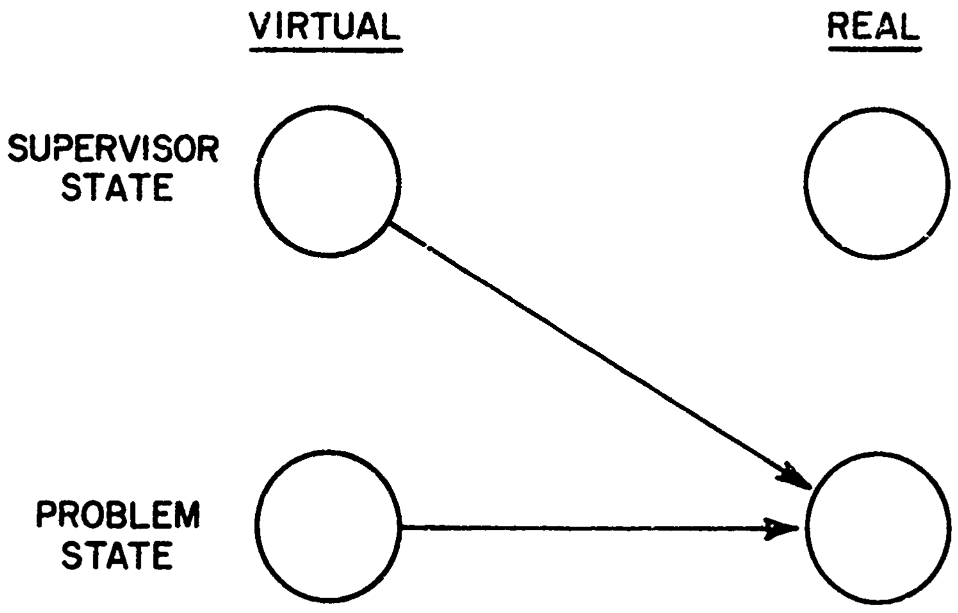
Denning [37] has recently delineated the extent of III generation systems by uncovering the existence of a "Late Third Generation." Since his distinction between III and Late III is often based on facilities of the operating system software, we need not distinguish them. Virtualization is concerned with the software visibility of the interior decor, not of any operating system features. Later in Chapter 4, we take up the likely characteristics of IV generation architectures. Since many of these same "Late Third Generation" facilities, e.g., interprocess communication, are now present in the IV generation interior decor, virtualization must consider them. [See Section 4.1.]

3.2 EMPIRICAL HARDWARE REQUIREMENTS

CP-67 has suggested a general approach for constructing virtual machines on other conventional III generation systems. In this section we will explore the application of the CP-67 virtual machine construction technique and see for which III generation systems it is appropriate. Appendix A provides a tutorial on CP-67 and describes the maps which it employs. The key point in the simulation of a (III generation) virtual processor is that since the host and virtual machine are identical (or similar), the instructions of the virtual machine may be executed directly on the host. However, because of the virtual machine mapping construction it is necessary to prevent certain of the instructions from executing directly on the host. These instructions, if executed directly by a VM can cause serious difficulties in interpretation or control. [See Appendix A.] Any instruction whose direct execution cannot be tolerated is termed a _sensitive_ instruction. Then the key to implementing a virtual machine on III generation systems is to provide complete functional equivalence with a real machine without allowing the direct execution of sensitive instructions.

The scheme adopted relies on a software mapping and permits only the VMM to run in supervisor state. The virtual machine, itself, is run in problem state. Therefore, if the instructions that are considered sensitive are also privileged, their attempted execution by the virtual machine (in problem state) will cause a trap to the VMM. If there are some instructions that are considered sensitive but are not privileged instructions, then it may be necessary to resort to massive interpretation in order to preserve the integrity of the virtual machine.

What are the instructions that are considered sensitive in a virtual machine system? Certainly any instruction that attempts to change the mode of the virtual machine. As stated above, in order to prevent a virtual machine from gaining control of the host machine, the virtual machine must be run in problem state. Moreover the virtual machine should not be permitted to put the host machine into supervisor state in such a way as to allow the virtual machine to execute any privileged instructions. In order to maintain a compatibility between the virtual machine and its real counterpart, the VMM must provide a virtual supervisor state. See Figure 3-1. In this state, the virtual machine's privileged instructions are simulated by the VMM. In addition, the virtual machine should not be able to change its state from virtual supervisor to virtual problem state without a privileged instruction. Since both of these states are actually run on the host machine as problem state, the VMM must be informed directly or it would have no way of knowing that a virtual state change has occurred. Furthermore, it is not sufficient to merely prevent the virtual machine from changing the state of the host machine, it must also be prevented from referencing it. Since the host machine will be in problem state when the virtual machine thinks it is in supervisor state, the answer to the question "What state am I in?" may give an inappropriate result.



III GENERATION VIRTUAL PROCESSOR MAP

FIGURE 3-1

Consequently the instructions which reference state information as well as those that actually change it must be privileged instructions. In machines in which this is not true, if the VMM cannot perform some kind of pre-interpretation, a virtual state change might occur without its knowledge. As will be seen in Section 3.3, the hybrid virtual machine, HVM, provides another solution to this dilemma.

Another problem arising from the fact that virtual machines in virtual supervisor state must, in fact, be run in physical problem state, concerns the interpretation or lack of interpretation of certain bits in the instruction wo.[illegible]. Certain machines use an additional bit in the instruction word only when in supervisor state. Other machines use additional bits in the address portion of the instruction. These differences in instruction execution in problem and supervisor state may be difficult to resolve without resorting to instruction-by-instruction simulation. [See Appendix B.]

In addition to changing or referencing the sta[illegible]e of the host machine, the virtual machine must be prevented from tampering with or looking at certain sensitive registers and core locations. For example, because the VMM must keep track of the real interval timer for all machines, the virtual machine must be prevented from referencing the real clock. Since in some machines the clock is a register or a fixed location in core, either the instruction that references it must be privileged or some method must be found for protecting it through the use of the protection or relocation system. In order to preserve the virtual machine notion, the VMM must maintain a virtual clock for the virtual machine in some other part of core. Some other sensitive dedicated core locations are the various interrupt registers that are automatically accessed by the hardware in the event of an interrupt. Since the contents of the interrupt locations are used automatically by the hardware, their values may be considered part of the specification of the state of the machine. Therefore, the instructions that access these values are sensitive and should be privileged. If they are not, some other means for protection and alerting must be employed.

Other sensitive instructions are those involving the storage protection system and address relocation system. In order to protect the supervisor and (possibly) other virtual machines from an errant virtual machine, the virtual machine may never have access to any location that is not in its virtual memory. This may be accomplished by making all storage protection instructions privileged, or permitting their effect only within the address space defined by the address relocation system. If there is no relocation system and absolute addresses are generated, then the storage protection mechanism must be used to set the virtual memory size, protect dedicated lower core locations from access, and protect the supervisor. In this case, the storage protection instruction must be privileged to permit the supervisor to manage the protection keys for the host machine. If the host machine has a storage relocation system (either relocation registers, paging, segmentation, or a combination of these) then it must be insulated from the action of the virtual machine by making all instructions referencing the relocation system privileged. If it is desired to have a virtual relocation system which makes use of the host machine's relocation hardware then the effect of relocation setting instructions must be simulated. This simulation includes translating virtual memory locations to real core addresses.

Input-output instructions are always sensitive. It is necessary that they be executed only by the supervisor in order to isolate core from secondary storage, protect secondary storage from virtual machines, and enable the efficient scheduling of I/O tasks for the host machine. Furthermore, the VMM may be required to translate virtual device addresses into their real equivalents before I/O transmission may take place. If the notification of the I/O instruction does not occur before the I/O operation is started, data may be lost or physical movement may be needlessly initiated. Thus it is important to trap the I/O instruction as soon as possible. A privileged I/O instruction is the best solution.

Thus, we are able to state the following empirical hardware rules which govern whether a virtual machine monitor may be implemented on a conventional third generation computer system.

Empirical Hardware Requirements

(1) The method of instruction execution of non-privileged instructions in both supervisor and problem state must be roughly equivalent for a large subset of the instruction repertoire.

(2) A method of protecting the supervisor (and any other virtual machine, if there is multiprogramming) from the active virtual machine must be available. This may be accomplished, for example, through a protection system or an address translation system.

(3) A method of automatically signalling the supervisor when the virtual machine attempts to execute a sensitive instruction must be available. The trap must not cause unrecoverable errors. It must then be possible for the supervisor to simulate the effect of the instruction. Sensitive instructions include:

a. Those instructions which alter or query the state of the machine, e.g. "Is it in supervisor state or problem state?", "Is it in relocate mode or not?"

b. Those instructions which alter or query the state of the machine's reserved registers and core locations.

c. Those instructions which reference the storage protection mechanism, the memory system, or anything else that is specifically used by the VMM in building and managing the virtual machine.

d. Any I/O instruction.

Application of these hardware rules to a number of familiar computer systems is treated in Appendix B. The results of these case studies are summarized in the table of Figure 3-2. Except for the IBM 360 family of machines, most other contemporary III generation systems are not virtualizable.

Machine	Hardware Rule Violations
IBM 360/67	none
IBM 360/65	none
HITAC 8400	none
IBM 360/85	none
DGC NOVA	none
DDP-516	3, 3a, 3b
GE 635	3a, 3c
GE 655	3a
Multidata A	3
XDS 940	1
PDP-10	3a
BBN TENEX	3a

Note: In some case, a machine which violates hardware virtualization rules may still be able to support an HVM. [See Section 3.3.]

III GENERATION VIRTUAL MACHINE CASE STUDIES-SUMMARY
FIGURE 3-2

3.3 HYBRID VIRTUAL MACHINES AND REQUIREMENTS

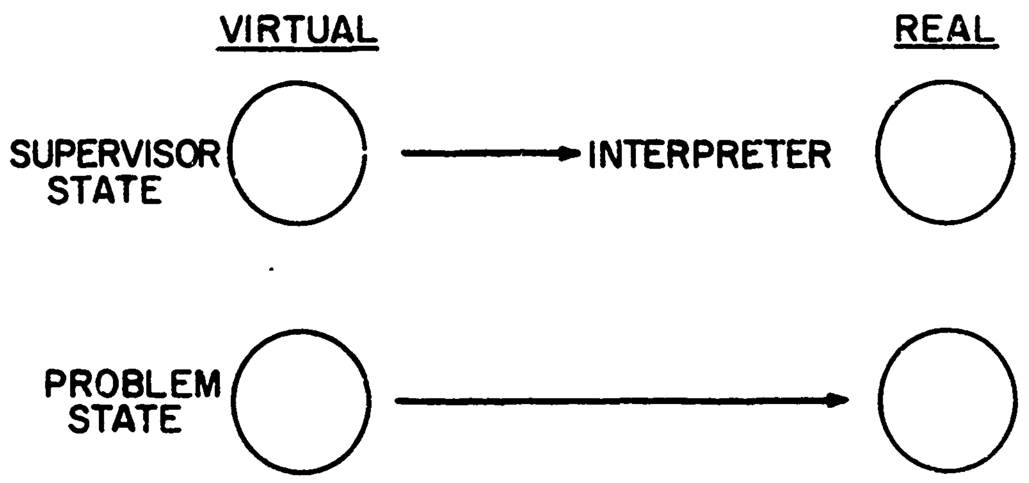
As we have illustrated in Figure 3-2 [also Appendix B] most III generation machines .e not virtualizable because of violations of the empirical rules. Thus, it is desirable to define another construct which has the advantages of virtual machines, e.g. functional equivalence to real machines, but is feasible on a larger class of systems then (conventional) virtual machines are. Furthermore this construct must be accomplished without the disadvantages of the complete software interpreter machine. The hybric virtual machine, HVM, defined in Chapter 2 meets both of these objectives.

An HVM is functionally equivalent to a real machine. All instructions issued within the most privileged layer of the HVM are software interpreted while all non-privileged-layer instructions execute directly. This leads to an implementation on III generation syste , in which virtual problem state is mapped into physical problem state but, unlike the conventional III generation virtual machine, the HVM virtual supervisor state is not also mapped into physical problem state. See Figure 3-3.

Without mode-mapping and direct execution of the non-sensitive supervisor state code (as in the conventional III generation VCS), the HVM streamlines the empirical hardware requirements. No longer need one be concerned that all non-privileged instructions are insensitive to the mode map. Thus, the HVM eliminates empirical Rule 1 from its set of hardware requirements.

Attempts by the virtual machine to enter supervisor state are directed to the VMM. Since the VMM maintains the virtual supervisor state as pure software construct, a weaker form of Rule 3a is all that is needed for HVM. In particular, we must identify only those instructions which enter supervisor state. Thus, since supervisor state is not mapped, a mode query need not be either and the instruction is not sensitive. Finally, since execution in virtual supervisor state is under the control of an instruction-by-instruction interpreter, the return to problem state instruction is not sensitive. The interpreter itself can read the instruction and observe that a virtual state change has occurred.

As illustrated in Figure 3-2 [also Appendix B], a number of III generation machines fail the virtual machine empirical hardware requirements either because of Rule 1 or Rule 3a. Thus, they are candidates for an HVM.



HVM MODE MAPPING

FIGURE 3-3

3.4 AD HOC HARDWARE "IMPROVEMENTS"

In addition to the HVM, another approach to constructing virtual machines on third generation architectures might involve proposing some ad hoc "improvements" to permit more systems to satisfy the empirical requirements (of Section 3.2). We term these improvements "ad hoc" since they respond to specific hardware deficiencies which arise in the use of the conventional III generation software virtual machine techniques. Only in Chapter 4 do we consider more major architectural changes that follow from a different view of virtual machines.

The ad hoc improvements we explore are addressed predominantly to machines which violate empirical rule 1 or (one or more subparts of) empirical rule 3 [Section 3.2]. We will assume that the machines discussed in this section do not trap so as to cause unrecoverable errors. Thus, for example, we will not examine the DDP-516 which traps an entire instruction late [Appendix B]. Rather, the difficulties we consider are either machines in which (1) Sensitive instructions do not trap when necessary, or (2) Undesirable side effects are caused in order to guarantee some trap.

An example of difficulty 1 is a machine in which some sensitive instruction is not privileged and so does not trap when execution is attempted in problem state. The PDP-10 JRSTF instruction is such an example [Appendix B]. Another case of difficulty 1 is a machine in which privileged instructions execute as a NOP in problem state. The Multidata Model A or GE

535 demonstrate this difficulty [Appendix B].

An example of difficulty 2 is a machine in which many "innocent bystander" core locations must be protected in order to be able to protect some other sensitive location because the machine's relocation or protection mechanism is too coarse or too inflexible. The 360/65 or HITAC 8400 must protect an entire 2K byte core block just in order to make several hundred sensitive locations inaccessible [Appendix B].

3.4.1 Difficulty One-- Non-trapping Sensitive Instructions

Difficulty one may be restated as a problem: Choose the set of privileged instructions so as to include all sensitive instructions which may not be "protected" by some other mechanism, e.g. protection or relocation system. The solution to this problem must be formulated in such a way that if, during the course of VMM design, it is discovered that an unanticipated instruction is sensitive, then the architecture will still be virtualizable.

A valid solution is to permit the supervisor to make any instruction privileged by executing a special Trap On Opcode (TOO) instruction. We assume that initially there are no privileged instructions and that the VMM is in control. The VMM initializes the system by executing a TOO instruction for each opcode, including TOO, that is to be made privileged. At the time of machine design, there is no need to make a commitment on which instructions are to be privileged. When the VMM has been written (and debugged) and it is known which instructions are sensitive, then the appropriate initialization may be added to make those instructions privileged. Furthermore, if the computer system is to be used for non-virtual machine tasks, it may be desirable to return certain instructions to a non-privileged state.

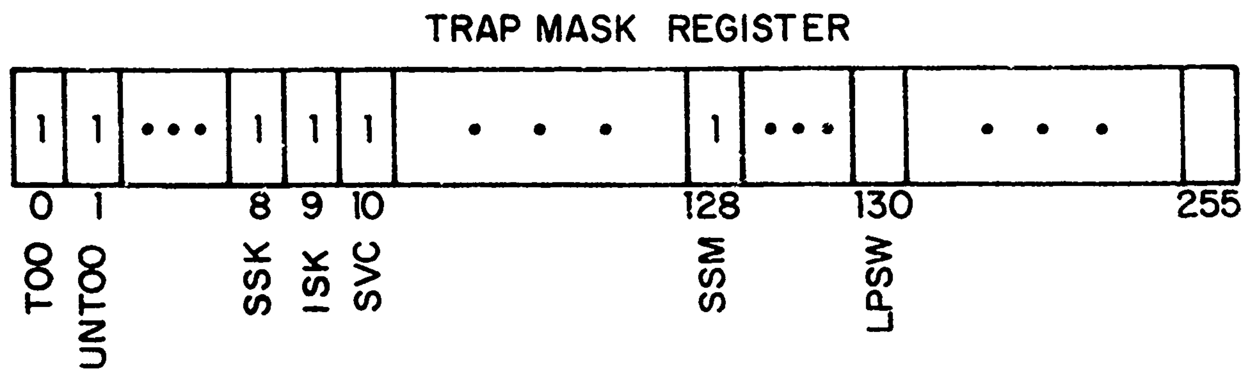
It is necessary to show that the TOO instruction has the desired effect and that, furthermore, the effect of TOO instructions may be simulated for virtual machines. For the purposes of this discussion, we can assume that an opcode is the operand of a TOO instruction.

Thus, there are two new instructions introduced into the computer:

> TOO-- described above
>
> UNTOO-- which turns off trapping for the operand opcode and sets a condition code indicating whether trapping was previously on.

Figure 3-4 gives a simplified example (of one possible implementation) in connection with the IBM 360. The operand field is assumed to be m bits long. The TRAP register is 2**m bits. [The trap register is shown as a bit mask. This is just one possible implementation. For example, it could be an associative memory.] Execution of the instruction "TOO o", where o is a valid opcode, causes bit o of the TRAP mask to be set to 1. Subsequent execution (in problem state) of any instruction whose bit is turned on in the trap register causes a trap to occur. Thus, sensitive instructions may be made privileged.

The action of the TOO and UNTOO instructions may be



```
TOO    TOO
TOO    UNTOO
TOO    SSK
TOO    ISK
TOO    SVK
TOO    SSM
TOO    LPSW
```

Initialization sequence for modified 360 with trap register.—Status before 'TOO LPSW' instruction is executed.

Instruction is privileged if bit set in trap register.

TRAP REGISTER AND TOO INSTRUCTION

FIGURE 3-4

simulated for virtual machines. The VMM keeps a copy of the virtual machine's trap register, i.e. virtual trap register. Attempted execution, by the virtual machine (running in problem state, as before) of the TOO or UNTOO instruction causes a trap to the VMM. The supervisor then simulates the effect on the virtual trap register. Before the virtual machine is dispatched in (virtual) problem state, the virtual trap register is OR'ed into the (real) trap register. Figure 3-5 illustrates how the virtual trap register is supported.

3.4.2 Difficulty Two-- Undesirable Protection Side-effects

The problem considered here is largely one of inelegance, awkwardness, or unreasonable performance degradation caused by the side effects of mechanisms required in order to limit access to certain sensitive (core) locations. Most computer systems have a number of sensitive locations, often termed the interrupt control area [69]. Since the interrupt control area is located in physical core, it may be tampered with by non-privileged instructions. Consequently, it is necessary (as brought out in Section 3.2) to protect these physical locations by means of the protection or relocation hardware of the host. Unfortunately, the protection or relocation system is often too "coarse" to perform this task neatly and, as a result, other non-sensitive locations may be protected as well.

There are roughly two points of view that may be adopted to avoid difficulty two. In the first solution, we alter the machine design to provide a finer grain protection mechanism.

TR_{vm} --- VM trap register

TR_{vmm} --- VMM trap register

TR_s --- traps for sensitive instructions

to run virtual machine ----

$$TR_{vmm} := TR_{vm} \vee TR_s$$

trap on opcode p ---

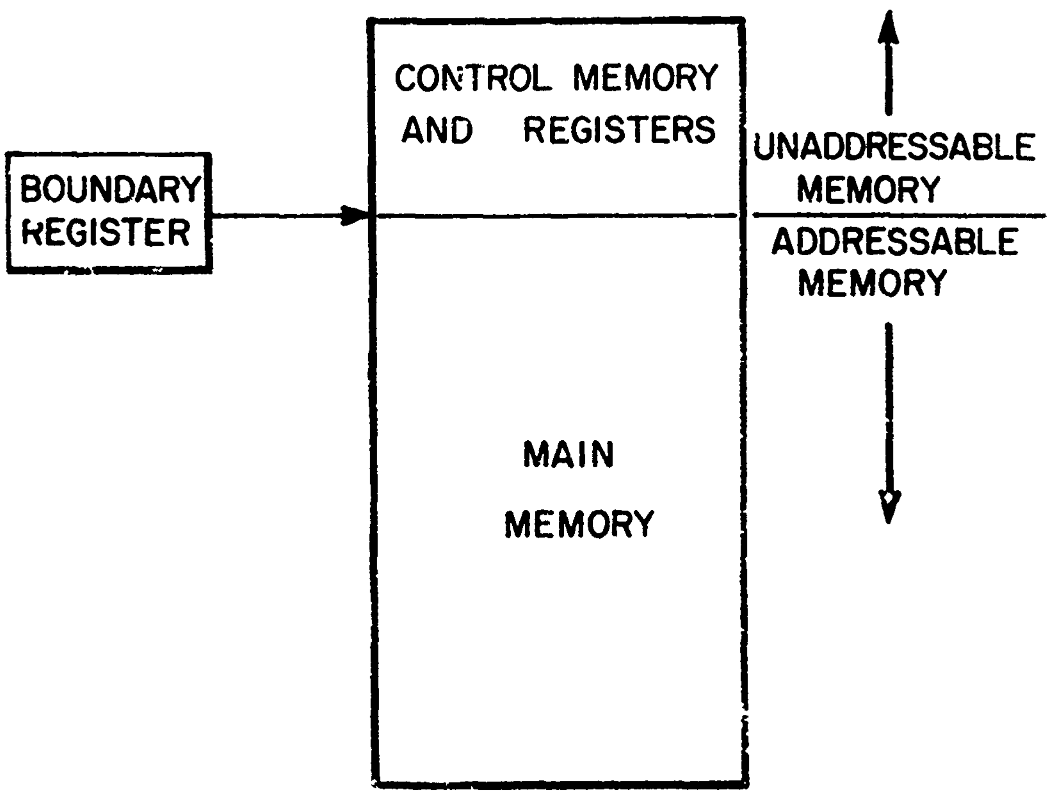
	$p \in TR_s$	$p \notin TR_s$
$p \in TR_{vm}$	SIMULATE TRAP	SIMULATE TRAP
$p \notin TR_{vm}$	SIMULATE INSTRUCTION	EXECUTE NORMALLY

VIRTUAL MACHINE SUPPORT FOR TRAP REGISTER

FIGURE 3-5

Instead of protecting core on the basis of 2K blocks (as in the 360/65), we can do it on a word basis. We introduce a special new instruction Trap On Location, TOL, which may be used analogously with TOC (above) to initialize the sensitive locations for a virtual machine. After a location has been TOLed, an attempt to reference it in problem state causes a trap. There are a number of different ways in which TOL may be implemented. For example, an associative memory can be used to hold locations that have been TOLed. Or, each word of memory may be associated with one bit which indicates if it causes a trap on reference. This second implementation has been used in the design of the ISPL machine at the RAND Corporation [11,12]. The first implementation with a one word (associative) memory has been used in the IBM M44 [91] and the MIT Artificial Intelligence Laboratory PDP-10 [61,62] to simulate address stop switches.

In the second solution, we remove the sensitive locations from main memory and place them in some unaddressable or differently addressable memory. In order to access this special memory, we require a new privileged instruction. See Figure 3-6. In some machines, e.g. 360/67, 370, this type of auxiliary memory is sometimes called control memory or control registers. There are several good reasons for adopting this solution. First associating sensitive locations with privileged instructions which access them provides a homogeneous mechanism for virtual machine trapping. Another reason for adopting solution two is that some of the locations in the interrupt control area are an intimate part of the description of the state of the processor.



SENSITIVE LOCATIONS IN UNADDRESSABLE MEMORY

FIGURE 3-6

Thus, they should be associated with the processor (on a more individual basis) as the PSW is.

Solution two as an ad hoc improvement was recognized by Fuchi et al [50] in their work with a virtual machine for the HITAC 8400.

> We have two proposals for further hardware improvments. One is that memory protection should become more flexible to include read protection. The other is that the CAW storage area and the timer should be moved from the main core memory to the scratchpad memory. The latter takes some trouble in simulator construction and is inelegant from the logical designer's point of view [50].
>
> If all of these requirements are met, we could build a more natural simulator ... [50].

As mentioned above, the System/370 incorporates some aspects of solution two. Some of the new, sensitive locations have been made "control registers" and are accessible only via privileged instructions. The new channel and CPU identification numbers are invisible and accessible only via privileged instructions. The new 370 clock is invisible and can be stored into only with a privileged instruction. Unfortunately, it may be read by a non-privileged instruction.

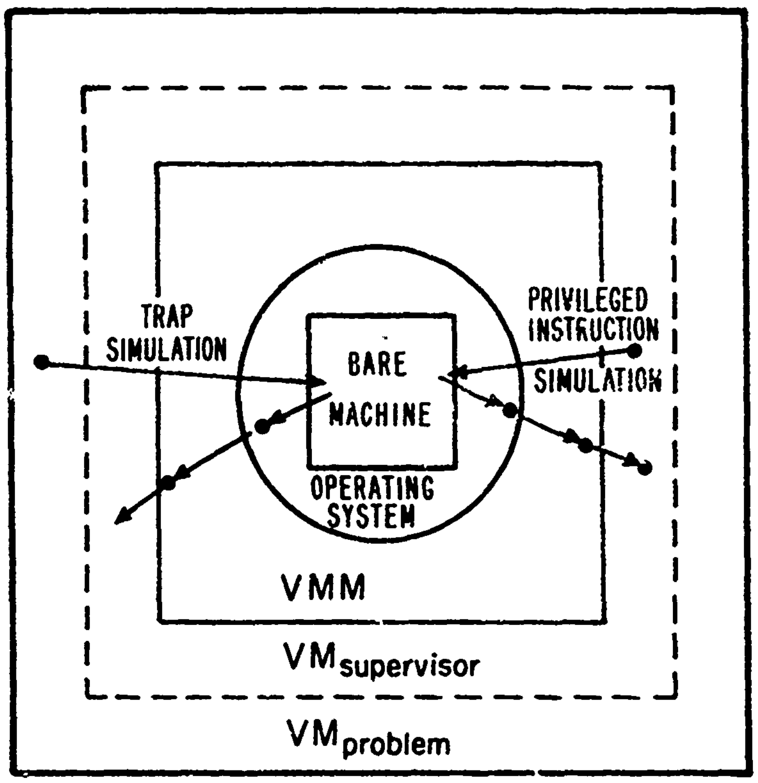
3.5 III GENERATION EMPIRICAL SOFTWARE REQUIREMENTS

In Section 3.2, we developed the empirical hardware requirements for Type I virtual machines. In this section, we add the empirical software requirements that must be introduced for Type II (extended machine host) virtual machines. Recall that a Type II virtual machine system [Chapter 2] is one in which the virtual machine monitor runs under a host operating system rather than on a bare hardware host machine.

Thus in Figure 3-7, hardware events such as the trap caused by an attempt to execute a privileged instruction in the supervisor state of the virtual machine [physical problem state], get passed to the host operating system. This operating system then gives control to the VMM (under it) which may perform privileged instruction simulation for the virtual machine.

A Type II virtual machine may be desirable in a number of circumstances as pointed out in Chapter 2. Because a Type II virtual machine supervisor has a (possibly) rich set of supervisory services (SVC's) available to it courtesy of the host operating system, it often requires less effort to implement a Type II system than a Type I system. At the same time, however, the SVC's and the SVC mechanism must be of a form that does not make it impossible to implement a Type II virtual machine.

Thus the dilemma. The host operating system must provide those supervisory services (primitives) that make efficient the implementation of a Type II virtual computer system, without making it impossible.



TYPE II VMM

FIGURE 3-7

Since the empirical software requirements will complement the empirical hardware requirements, we begin our search with another look at the requirements of Section 3.2. Empirical hardware rule 1 still remains valid. The introduction of the additional structure ' an operating system between the virtual machine, VMM, and the hardware must do nothing to invalidate rule 1.

The Type II system version of rule 2 is that there must be a primitive available to protect the virtual machine monitor from the active virtual machine. Since the virtual machine monitor will be running in an insulated environment that, likely, will prevent it from utilizing the protection or relocation hardware directly, there must be a system primitive that it can invoke to obtain the desired effect.

Rule 3 must be modified to indicate that when the virtual machine traps, the operating system supervisor directs the signal to the VMM for processing. Since the operating system and VMM are no longer a single unit, there must be a method of communicating to the operating system supervisor that the virtual machine is a particular subprocess of the VMM and that all of the appropriate virtual machine traps are to be directed to the VMM. The mechanism for setting up the subprocess and indicating where the signal is to go may be accomplished either by a formal subprocess primitive, or it may be an informal procedure that is made up of a number of the other primitives. It is not sufficient for the virtual machine to merely signal the VMM. As before, the trap from virtual machine to VMM (now, by way of the operating system) must not cause unrecoverable errors. It must be possible for the VMM to simulate the effect of the operation. Thus, it is not satisfactory to set up a subprocess in which anything that appears anomalous is considered to be an error and the subprocess is destroyed. This is actually fairly common among systems.

Since primitives (SVC's) alter or query the state of the machine, they are sensitive according to hardware requirement 3a. Thus, their attempted execution by the virtual machine is forbidden and must cause a trap to the virtual machine supervisor. Consequently, the operating system requires a primitive that may be used to turn on or off the SVC's for a subprocess. In the case of a virtual machine, the VMM would use this primitive to shut-off all SVC's. This, of course, includes the SVC's to turn on and off the SVC's.

Note that in a well designed 'user machine' [42,75], system primitives replace the hardware instructions to accomplish the sensitive instructions a,b,c, and d of Section 3.2. Thus, shutting off the SVC's makes these sensitive instructions (along with all other SVC's) trap in the 'user machine.'

Thus, we are able to state the following empirical software rules which govern whether a Type II (extended machine host) VCS may be implemented under the operating system of a virtualizable third generation computer system.

Empirical Software Requirements

(S1) The method of instruction execution of non-privileged instructions in both supervisor and problem state must remain equivalent for a large subset of the instruction repertoire.

(S2) A primitive for protecting the virtual machine monitor (and any other virtual machine if there is multiprogramming under this VMM) from the active virtual machine must be available. This may be accomplished, for example, through a protection primitive, address translation primitive, or some more general subprocess primitive.

(S3) A primitive to tell the operating system supervisor to signal the virtual machine monitor when the virtual machine attempts to execute a sensitive instruction must be available. The signal to the VMM must not cause unrecoverable errors. It must then be possible for the virtual machine monitor to simulate the effect of the instruction. The sensitive instructions are the same as for the hardware requirements.

Application of these software rules to several computer systems is treated in Appendix C. The results of these case studies are summarized in Figure 3-8. Except for the ITS, Incompatible Timesharing System, which provided the necessary primitives, the operating systems examined either failed the empirical rules or required ad hoc or specialized solutions.

Operating System	Software Rule Difficulties
HITAC 8400 TOS/TDOS	Ad hoc solution ($S_2 + S_3$)
IBM 360/67 UMMPS	Specialized solution ($S_2 + S_3$) (SVC SWPTRA)
IBM 360 OS/360	Violates rules ($S_2 + S_3$)
DEC PDP-10 ITS	Satisfies rules
IBM 360 CMS	Specialized solutions are possible.

III GENERATION TYPE II CASE STUDIES-SUMMARY

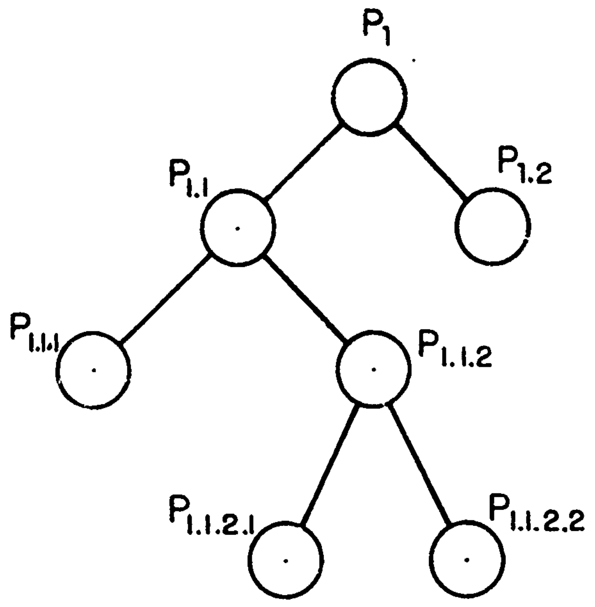
FIGURE 3-8

3.6 SOFTWARE PRIMITIVES FOR TYPE II VCS'S

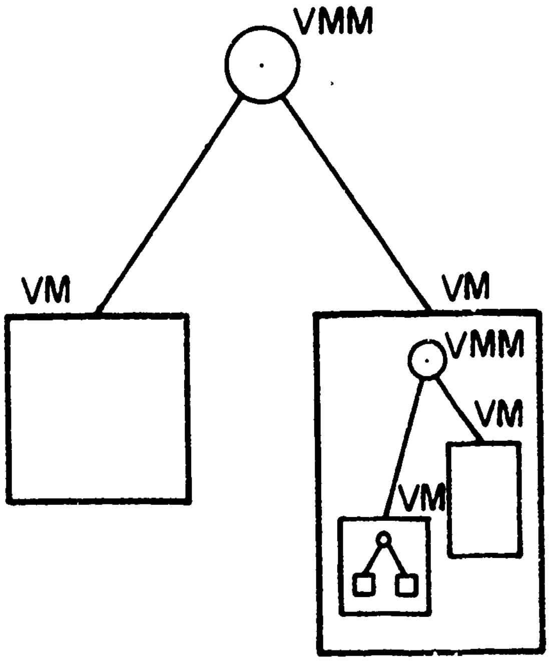
As we have illustrated in Figure 3-8 [also Appendix C], most III generation operating systems are not designed to support Type II VCS's. The systems which do have largely been reworked to do so. From the experience with the ITS Type II HVM [Appendix C] it becomes possible to indicate the kind of operating system primitives that would simplify the support of Type II third generation virtual machines. To avoid the usual problems with asynchronous I/O, we will discuss the primitives only with respect to the CPU and memory of the virtual machine.

The ITS UUO's, for example, provide facilities slightly different from those needed in supporting virtual machines. Since the goal of the ITS primitives is to support an extended machine environment, the primitives provide greater flexibility than is needed for virtual machines. For example, in ITS a superior may destroy all of the inferiors in the sub-tree below it. In a virtual machine environment, the VMM need only be able to destroy an immediate inferior, since that is the extent of the sub-tree. Support of additional levels of virtual machines is transparent to the VMM and resides with the second copy of ITS and the second VMM which are running in the level one virtual machine. The copy of ITS in the real machine does not know about the level two virtual machine and only sees a level one machine. Hence, the primitive is more general than is specifically needed for VMM construction. See Figure 3-9.

Other primitives, such as the Dennis-Van Horn primitives



Typical Multi-level process tree



Type II Two-level Virtual Machine Tree.

The VMM only sees one level beneath it.

PROCESS TREES
FIGURE 3-9

[40] and Lampson [74], are concerned with supporting dynamically changing relationships between processes. This feature may not be necessary with virtual machines.

On the other hand, operating system primitives usually do not directly support such intimate hardware features as the relocation system. Thus, for example UMMPS had to introduce SWPTRA to switch segment numbers.

A set of possible virtual machine primitives is proposed below. The intent of this set is to indicate the simplicity of a VMM that might be programmed using them. This example omits the consideration of I/O or time.

(1) i:=virtualmachine loc,j,k, ... ;

This primitive creates an inferior virtual machine process. The parameter loc indicates the location to transfer to or a trap from an inferior and i is the virtual machine's identifier. The other parameters might indicate virtual memory size, relocation or protection information, virtual processor instruction counter and register values, maximum quantum, etc.

(2) dispatch i;

Activate or reactivate suspended virtual machine i. Apply VMmap to derive physical register values from virtual values. VMM goes to sleep.

(3) fetch status i,j,w;

Obtain virtual register or memory location j from virtual machine i and write into word w of the VMM.

(4) <u>set status</u> i,j,w;

Write contents of VMM word w into virtual register or memory location j in virtual machine i.

(5) <u>destroy</u> i;

Destroy virtual machine i. This permits the identifier i to be used again.

Given that the host machine satisfies the Type I hardware requirements, then a Type II VMM (non-HVM) may be programmed using these primitives.

```
vmm:   i:=virtualmachine loc, 256K, ... ;
       dispatch i;
loc:   fetch status i,j,w;
       (SIMULATE OFFENDING INSTRUCTION)
       set status i,j,w;
       dispatch i;
       end;
```

The amount of status fetching and setting and simulation of sensitive instructions varies according to the machine's complexity and the difficulty of virtualizing. In a III generation machine which is easily virtualizable, the implementation of the VMM sketched above can be reduced to a straightforward program.

3.7 CONCLUSION

The main result of Chapter 3 has been the substantiation of the unsuitability of most existing III generation systems for supporting virtual machines. We have shown that the architectural characteristics of III generation systems and the definition of virtual machine lead to a particular software construction of the virtual machine map. For example, the processor component of this software map takes the privileged layer of the virtual processor into the unprivileged layer of the real machine. This immediately leads to a set of empirical requirements, e.g. instruction execution must be insensitive to the software layer mapping, which distinguish III generation virtualizable architectures. A number of examples reveal the startling fact that most third generation architectures are not virtualizable. That CP-67, the best known virtual machine system, could be constructed on the IBM 360/67 was a happy accident.

In response to difficulties with virtualizing most III generation architectures we propose two alternatives. First, we introduce the hybrid virtual machine which has less stringent hardware requirements but potentially more overhead than the virtual machine. Then, we propose a number of simple ad hoc changes that might be made to III generation machines to simplify virtualization.

The rest of Chapter 3 is concerned with implementing a Type II (extended machine host) VMM on a third generation operating system. The virtual machine construction is basically the same as before. However, now the VMM must manipulate its components using the primitives of the host operating system. This leads to a series of empirical requirements, e.g. there must be an operating system primitive to shut off all SVC's of the virtual machine, that distinguish Type II virtualizable operating systems. Once, again, most third generation operating systems are not Type II virtualizable. The magnitude of this lack is pointed out by the ease of implementing a Type II VMM or an operating system that is virtualizable, e.g. the Type II HVM under MIT's ITS for the PDP-10 [Appendix C]. To further accentuate this fact, we propose a set of virtual machine primitives and show the ease of developing a Type II VMM using them.

However, the main point of Chapter 3 has been that no III generation systems (hardware or software) have been designed with the intention of supporting virtual machines. This has forced virtual machines to be introduced via a particular III generation software construction. Requirements imposed by this construction in turn, have disqualified almost all III generation machines from supporting virtual machines. Even the few existing systems, e.g., CP-67, require considerable software support which implies development and implementation costs, and sub-optimally efficient operation.

These consequences lead us to search for machine architectures, specifically intended to support VCS's. As noted in Chapter 1, these considerations gave rise to the proposals by

Lauer and Snow [78] and Gagliardi and Goldberg [51]. While the model of Chapter 4 is developed specifically in connection with providing for the orderly introduction of virtual machines into the IV generation, the principles which result are generally applicable to all architectures. Thus, by suitable interpretation of the model we can develop principles for III generation systems. This approach is illustrated in Section 4.9.

CHAPTER 4.

PRINCIPLES FOR IV GENERATION VIRTUAL COMPUTER SYSTEMS

The Virtual Machine Map and its Firmware Implementation

4.0 PLAN OF CHAPTER 4

In Chapter 4, we will discuss the problem of introducing virtual computer system capabilities into the IV generation. A recent paper [51] points out some of the inherent difficulties and complexities caused by likely IV generation architecture and argues for a hardware-firmware assisted virtualizer built into IV generation computer systems. In this chapter, we carry the notion of hardware virtualization somewhat further and provide a general framework for viewing virtualization of a IV generation computer system. Furthermore, the result that emerges is applicable to less complex architectures as well.

Section 4.1 introduces the likely characteristics of IV generation computer systems architecture. The critical architectural development of such systems will be a formalized firmware implementation of processes. A corresponding model for processes is introduced; the process map, ϕ, maps process names into resource names.

Section 4.2 develops a model for a IV generation VCS. We introduce the virtual machine map (or resource map) f which maps virtual resource names into real resource names. Then the virtual machine map is combined with the process map of Section

4.1. The composition of the two mappings, $f \circ \phi$, expresses the mapping from process names to real resource names, and thus expresses the mapping of names accessible to a process executing on a virtual machine. Other related notions, such as recursion and Type II virtual machines are introduced into the model. Some elementary properties of the maps and virtual systems are enumerated.

Section 4.3 uses the model of Section 4.2 to point out the unsuitability of a software implementation of the map f (as was studied in Chapter 3) for a IV generation VCS. The argument depends on the complexity of the process map, ϕ, and the likelihood of ϕ visibility by, say, the inner layer supervisory software. A specific example of the kind of difficulty a software implementation of f might cause is developed for protection rings.

Section 4.4 discusses the Venice Proposal, VP, introduced in "Virtualizeable Architectures" [51]. The VP provides an interim proposal for the design of a hardware-firmware assisted virtualizer (HV) for a IV generation VCS. The advantages and disadvantages of the VP are pointed out using the model of Section 4.2.

Section 4.5 introduces the design of a hardware-firmware virtualizer based directly upon the model of Section 4.2. The initial objective is automatic virtualization of CPU-Memory processes (with a minimum of software intervention). The new instructions introduced, modifications to the system base and expected performance are all indicated.

Section 4.6 provides a number of detailed examples to illustrate instruction executions on virtual computer systems constructed with representative hardware virtualizers.

Section 4.7 briefly considers some of the subtler problems raised earlier in the chapter. Proper virtual treatment of input/output, a technological difficulty, must await progress in the real treatment of input/output. In the interim, some ad hoc solutions to I/O, derived from III generation techniques, are provided.

Section 4.8 applies the model of Chapter 4 to a reconsideration of second and third generation computer systems. One design that emerges, the "$f=R-B$, ϕ=identity" machine, is similar to the machine recently proposed by Lauer and Snow [76]. The other design yields a modified IBM 360 with a greatly streamlined CP-67.

4.1 CHARACTERISTICS OF IV GENERATION COMPUTER SYSTEMS

The most distinctive architectural characteristic that will be found in IV generation machines will be the refinement and formal implementation in firmware of the process model [31,37,41]. According to this view, activities in a computing system will be performed by a number of cooperating sequential processes, each accomplishing some elementary task and communicating among themselves via a well-defined mechanism. Furthermore, the processes themselves will execute in an environment extended to include idealized objects, such as an idealized address space.

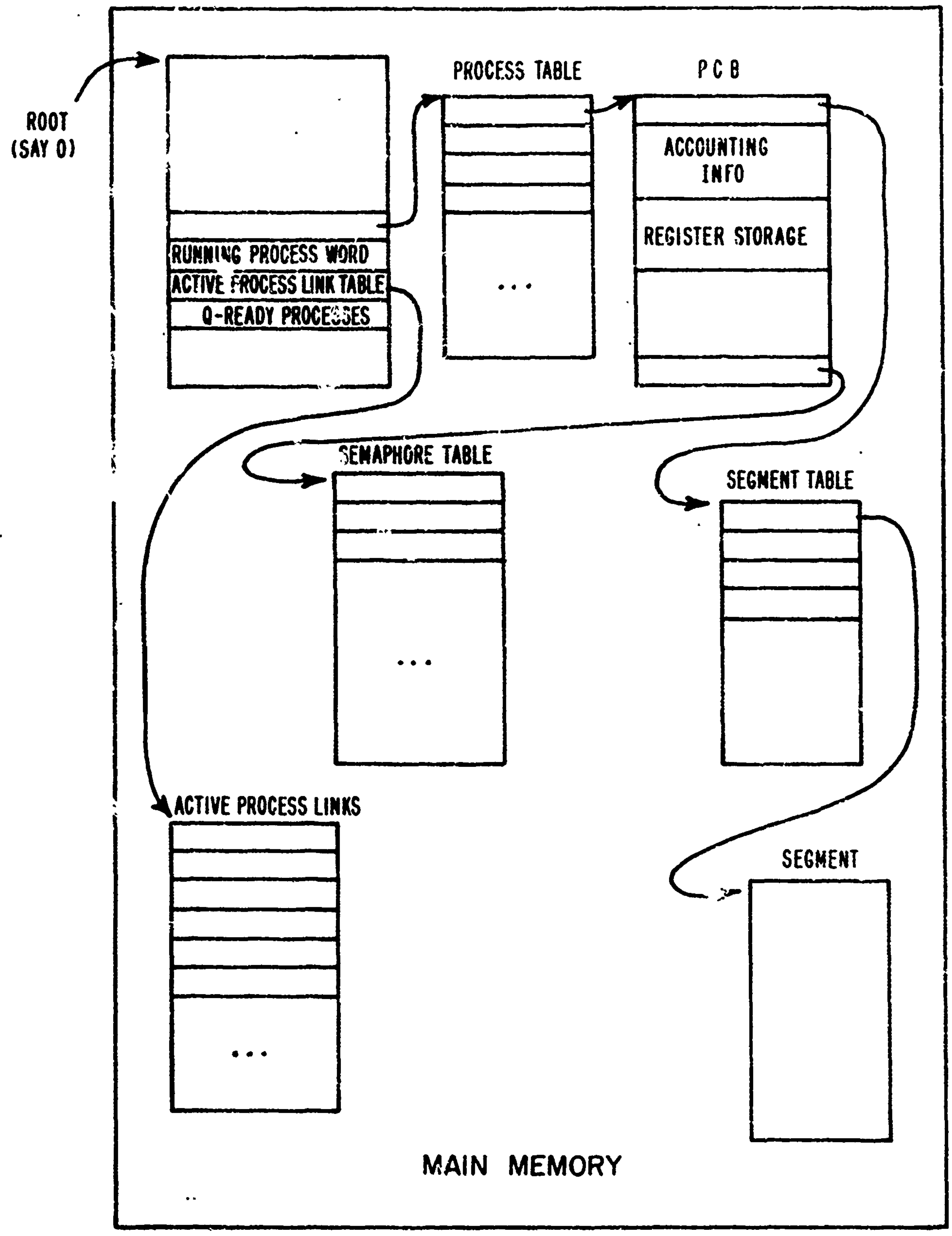
This view of the likely characteristics of IV generation architecture is based on the papers by G.M. and L.D. Amdahl [5,7], the Blauuw paper [19], the Morris paper [87], the Infotech book [70], and, particularly, on the existing Mitre Corporation VENUS system as described by Liskov [79]. Furthermore, we observe that key software features of generation n are often those features that are included in the interior decor of generation n+1. Therefore, Denning's recent tutorial [37] on (the software of) Late Third Generation systems strongly suggests that the process model will be incorporated into future systems.

Third generation computer systems rely on an ad hoc, purely software implementation of processes [37]. Thus, the choice of primitives, organization of tables, names of objects, etc., is not fixed and may be chosen by the software support. This implies that, for a given III generation system, there exists no unique implementation of processes.

By contrast, IV generation computer systems will feature a firmware implementation of processes. The complete collection of status information for all processes in the system will be called the system base [51,87]. The firmware will know the base location of the system base, the ROOT. Since tables and control blocks will be of fixed (pre-assigned) format, knowledge of the ROOT is sufficient for the firmware to locate any element in the system base. The system base will be located in main memory. For largely economic reasons [51,87], the system base will likely also be accessible to the inner layer, most privileged software [see below]. However, during process execution, the system base is referenced and updated directly by the firmware.

The system base will include elaborate process control blocks, queueing data bases, address space words, etc. [51,87]. See Figure 4-1. The queueing data bases will be used, for example, by the firmware dispatcher to invoke the READY process with highest priority [99]. The queueing data bases will also be used by the firmware in implementing interprocess communication via Dijkstra's P-operations, V-operations, and semaphores [41,79].

The process control blocks, (PCB's) will be pointed to by a process table whose location can be derived from the ROOT. Thus, an integer displacement in the process table, the process-id, identifies a particular process and the location of its corresponding PCB. The PCB will contain such information as temporary storage of the central processor's (CPU's) register



SYSTEM BASE

FIGURE 4-1

values, the priority of the process, and an address space word. The CPU will contain a process-id register whose value identifies the process in execution. Thus, there exists no notion of an absolute mode; execution is always on behalf of some process. Loading the process-id register activates the subject process and instructs the firmware to obtain register values, etc. from the process control block (PCB). In particular, the address space word is referenced and used to form the memory map. Thus, there exists no notion of an absolute addressing mode; all addresses are mapped.

The idealized address space will likely be a segmented layered structure [16,35,86,92,106]. Such a structure can be viewed as a generalization of the two layer Master/Slave modes of III generation systems. As the layer or ring [106] number of an active process decreases, the ability, by that process, to access segments in its segment table increases correspondingly. In ring 0, the most privileged layer, supervisory processes can also be expected to utilize a small set of necessarily privileged instructions. Privileged ring procedures of processes will also likely be invoked by process exceptions. This is the generalization of the trap to master mode in III generation systems.

The I/O systems of IV generation computer systems will likely be similar to those of the late III generation, e.g. IBM System/370. In particular, I/O-memory processes will very likely not be formalized to the extent that CPU-memory processes have been. I/O processor (channel) operations will execute in an absolute mode without process-id's or segmented addresses [8]. All address absolutizing, status maintenance, etc. will be performed by software in an ad hoc manner [70]. As a result of the lack of progress by IV generation designers in I/O process formalization and implementation, the IV generation virtual I/O considerations are largely the same as those of the III generation. Consequently, most of the discussion of Chapter 4 will omit I/O; only in Section 4.7 will we illustrate III generation techniques for introducing virtual I/O in IV generation systems.

Abstraction

A model of (CPU-memory) process structure may be obtained by a scrutiny of the system base. Execution by the CPU is always on behalf of some process. Hence, the names used by individual instructions are always process names, whether they are local or global (system-wide) names. For example, data or instruction addresses are always local names, defined by the segment table of the active process control block (PCB). Process-id's are global process names and their meaning is defined for all processes via a single table in the system base.

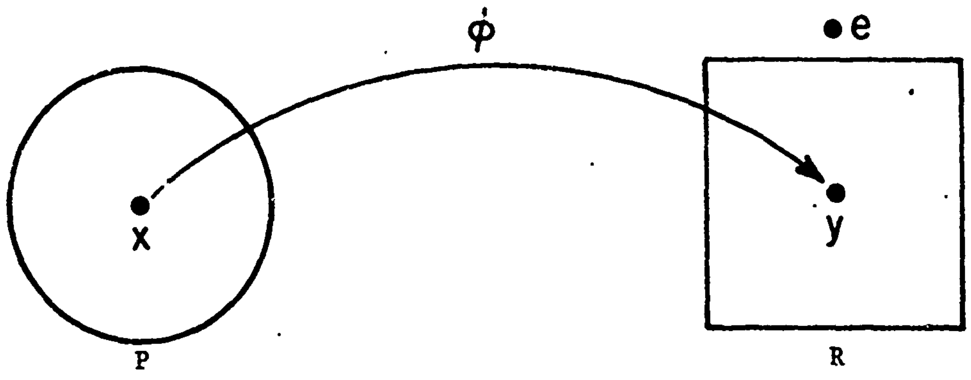
Effectively, then, the system base relates two classes of objects--those used by processes (whether local or global) and those related to the actual resources. The process names are idealized constructs while the resource names are determined by the hardware. There is no requirement that process names have the same form or structure as resource names. Thus, in particular, it becomes possible to speak of a process having a segmented, multi-dimensional address space. In contrast, the real (and virtual) memory resource address space must, of course, be linear to mirror physical construction. The distinction between derived and primitive resource names, such as a segmented and linear memory, is discussed in "Virtualizeable Architectures" [51].

We call the names used by processes, process names or P-names and those used by the hardware, resource names. In particular, the real resource names used by the physical hardware are called R-names. [However, as we shall see below, V-names are also resource names.] It will be necessary to discuss a mapping function, ϕ, which is implemented via the system base and maps process names into resource names. See Figure 4-2. [In Figure 4-2 and all figures that follow in Sections 4.1-4.4, process spaces are represented by circles and resource spaces by squares.]

Let the process space names be $P=\{p0,p1, \ldots, pa\}$. As discussed above, P includes the process memory space names and semaphore names (for the active process), all process-id's, etc. Let $R=\{r0,r1, \ldots, rb\}$ be the (real) resource space names. R also includes the processor and (at least conceptually) the I/O resource names, as well as the memory names. Then, for the active process, we provide a way of associating process names with (real/virtual) resource names during process execution. See

$\phi : P \rightarrow R \cup \{e\}$

$\phi(x) = \begin{cases} y & \text{if } y \text{ is the resource name for process name } x \\ e & \text{if } x \text{ does not have a corresponding resource} \end{cases}$



Examples :

$\phi(1|142) = 6142$
if seg 1 at location 6000

$\phi(12|3) = e$
if seg 12 does not exist

$\phi(\text{sem } 2) = 5000$
if sem 2's header at location 5000

$\phi(\text{process id } 40) = e$
if process 40 does not exist

THE PROCESS MAP ϕ

FIGURE 4-2

Figure 4-2. To this end, via the system base, we define, for each moment of time, a function

$$\phi: P \longrightarrow R \cup \{e\}$$

such that if $x \in P$, $y \in R$, then

$$\phi(x) = \begin{cases} y & \text{if } y \text{ is the resource name for process name } x \\ e & \text{if } x \text{ does not have a corresponding resource} \end{cases}$$

The value $\phi(x)=e$ causes an exception to occur to some exception handling procedure, presumably in an inner layer of this process on this machine. To avoid confusion with virtual machine faults [Section 4.2], process faults will always be called exceptions.

Section 4.6 provides detailed examples of the process map, ϕ. For clarity, a few simple examples follow. Throughout Chapter 4, we adopt the segmented address notation, s|d, where s is the segment number and d is the offset within the segment [86,92].

Examples: $\phi(1|142)=6142$ if segment 1 is located at 6000

$\phi(200|11)=e$ if seg 200 does not exist

$\phi(3|5000)=e$ if 5000 is outside bounds of segment 3

$\phi(\text{sem } 2)=5100$ if sem 2's header at location 5100 ϕ(sem 10)=e if sem 10 does not exist

ϕ(process-id 15)=400 if PCB for process 15 at loc 400

ϕ(process-id 40)=e if process 40 does not exist

4.2 MODEL OF A IV GENERATION VCS

In order to properly model the running of a process on a virtual machine, it is necessary to first clarify what is meant by a virtual machine and what are virtual resources. To do this, we formally introduce and identify an important new concept, the _virtual machine map_ (VMmap) or resource map. The VMmap expresses the relationship between the resources of a virtual machine and the resources of the real host machine. Consequently, the VMmap is a notion which exists completely independently of the process map or, for that matter, any software visible process structure. Separation and isolation of the resource map (VMmap) has often been confused or overlooked by previous authors [31,76,108], and this perhaps accounts for the lack of success in designing easy-to-program virtualizable architectures.

VMmap f

A resource name used by a virtual computer system will be called a virtual name, virtual resource name, or V-name, and the set of virtual names defines a virtual space or environment. A resource name used by the physical hardware is called a real name, real resource, or R-name, and the set of real names is called the real space or environment.

In the resource spaces, real and virtual, we must include the processor, memory, I/O system and any connections between them. For example, resource names must exist for every addressable unit (bit, byte, word, etc.) in the physical memory. Resource names must exist for every opcode of the processor. Names are also needed for all I/O devices.

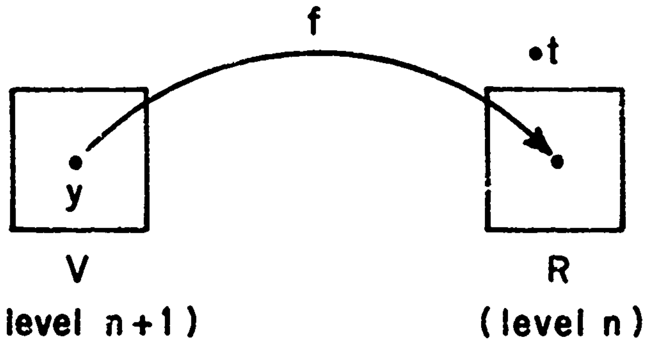
Therefore, the virtual computer system construction requires a mapping function that will map the complete set of virtual resource names into their corresponding physical resource names. This mapping function is illustrated in Figure 4-3. No a priori assumption is made about the nature of the mapping function f. [However, the implementation part of the VM definition [Section 2.1] implies that once the mapping is established most instructions of the virtual machine execute directly on the host.] The mapping function f is called the virtual machine map, resource map, or VMmap.

For future reference, we denote the virtual space names by V={v0,v1,...va} and the real space names by R={r0,r1,...rb}. V is the resource space of the virtual machine and R is the resource space of the host. We make no assumption, here, about the relative sizes of V and R. Both V and R include the memory space names. But they also include the other resources, as well. For example, we can imagine ordering the opcodes by assigning them the names following the memory space names in V and R. Lauer and Snow [76] have observed that it may be sufficient to consider only the memory space names since other resources are often coded to appear as memory. For example, they observe that the accumulators of the DEC PDP-10 and the I/O devices of the DEC PDP-11 are each addressable as memory locations.

Since we assume no a priori correspondence between virtual

$$f: V \longrightarrow R \cup \{t\}$$

$$f(y) = \begin{cases} z & \text{if } z \text{ is the real name for virtual name } y \\ t & \text{if } y \text{ does not have a corresponding real name} \end{cases}$$



Examples :

f (6142) = 12142

f (25000) = t if 25000 out of virtual memory

f (processor 3) = processor 2
if virtual processor 3 is real processor 2

f (processor 2) = t
if virtual processor 2 not assigned

THE VMmap f

FIGURE 4-3

and real names, we must incorporate a way of associating virtual names with real names during execution of the virtual machine. To this end, we define, for each moment of time, a function

$$f: V \text{---}> R \cup \{t\}$$

such that if $y \in V$ and $z \in R$ then

$$f(y) = \begin{cases} z & \text{if } z \text{ is the real name for virtual name } y \\ t & \text{if } y \text{ does not have a corresponding real name} \end{cases}$$

The value $f(y)=t$ causes a trap or fault to some fault handling procedure in the machine R. For clarity we always term a VM fault a fault, never an exception. The spaces V and R define two virtual machine levels. We say that the virtual machine V is at a higher level than the machine R. If the level of R is n, then the level of V is n+1. If P is the physical machine then it is level 0. Thus, a VM-fault is a VM-level fault and control passes to a fault handling machine at the next lower level. See Figure 4-4.

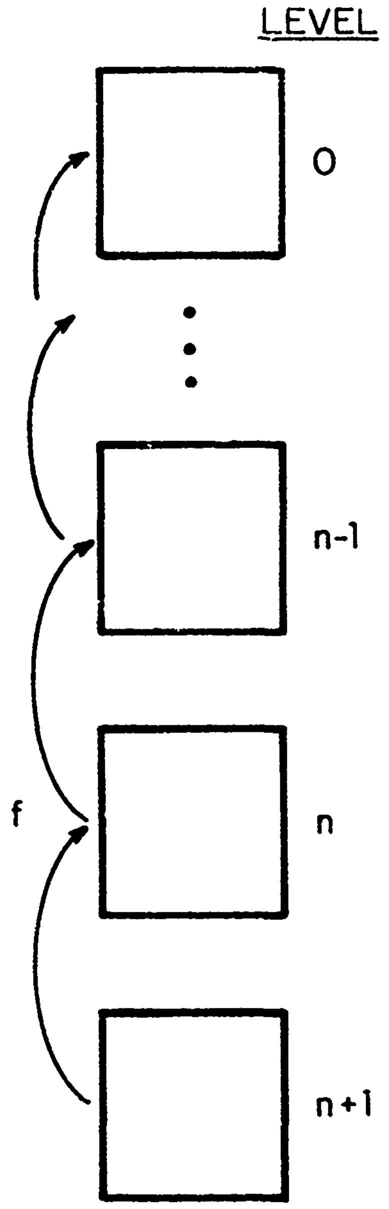
<u>Running a Virtual Machine</u>: $f \circ \phi$

In IV generation systems, running the virtual machine $V=\{v0,v1,...vb\}$ implies running some process $P=\{p0,p1...pa\}$ on the virtual machine. This defines the mapping

$$\phi: P \text{---}> V \cup \{e\}$$

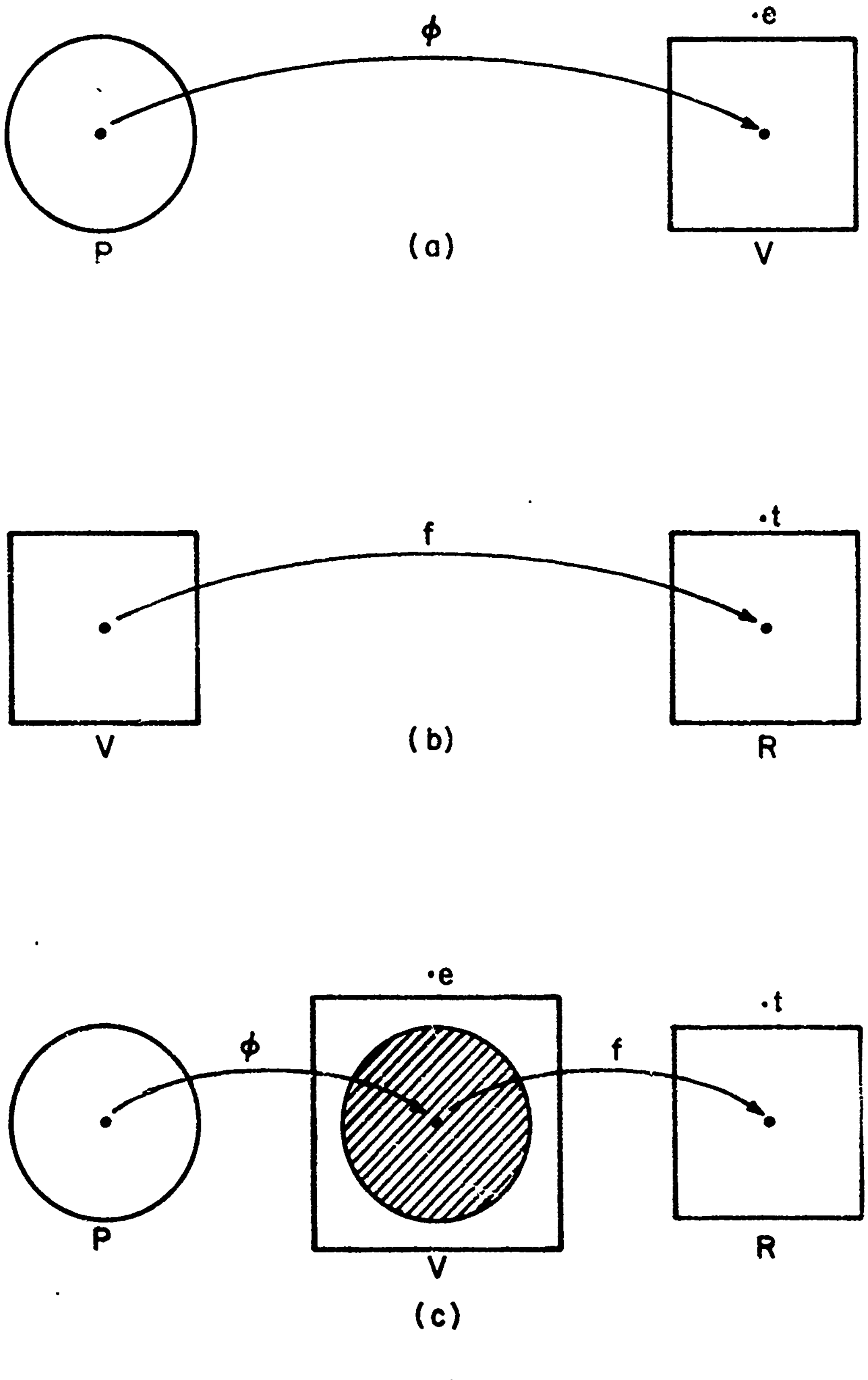
as before, with virtual resource names, V, substituted for real ones in the resource range of the map. See Figure 4-5a.

The virtual resource names, in turn, are mapped into their



VM LEVELS

FIGURE 4-4



fo φ
FIGURE 4-5

real equivalents by the map, $f: V \text{---}> R$. See Figure 4-5b. Thus, a process name x corresponds to a real resource $f(\phi(x))$. See Figure 4-5c. In general, process names are mapped into real resource names under the (composed) map

$$f \circ \phi: P \text{---}> R \cup \{t\} \cup \{e..$$

Section 4.6 provides examples which illustrate mapping process names under the composed map $f \circ \phi$.

f vs. ϕ

There is no a priori requirement that f or ϕ be of a particular form or that there be a fixed relationship between them. [However, in some sense, ϕ corresponds to how the users would like to program the machine while f corresponds to how the system would like to utilize its resources.] We discuss this point further in Section 4.8. We have indicated that in IV generation systems, the memory component of the map ϕ will likely be segmented. Since f maps resource spaces, its memory component cannot be segmented. The only requirement of f and ϕ is that the range of ϕ be included in the domain of f. (Figure 4-5c)

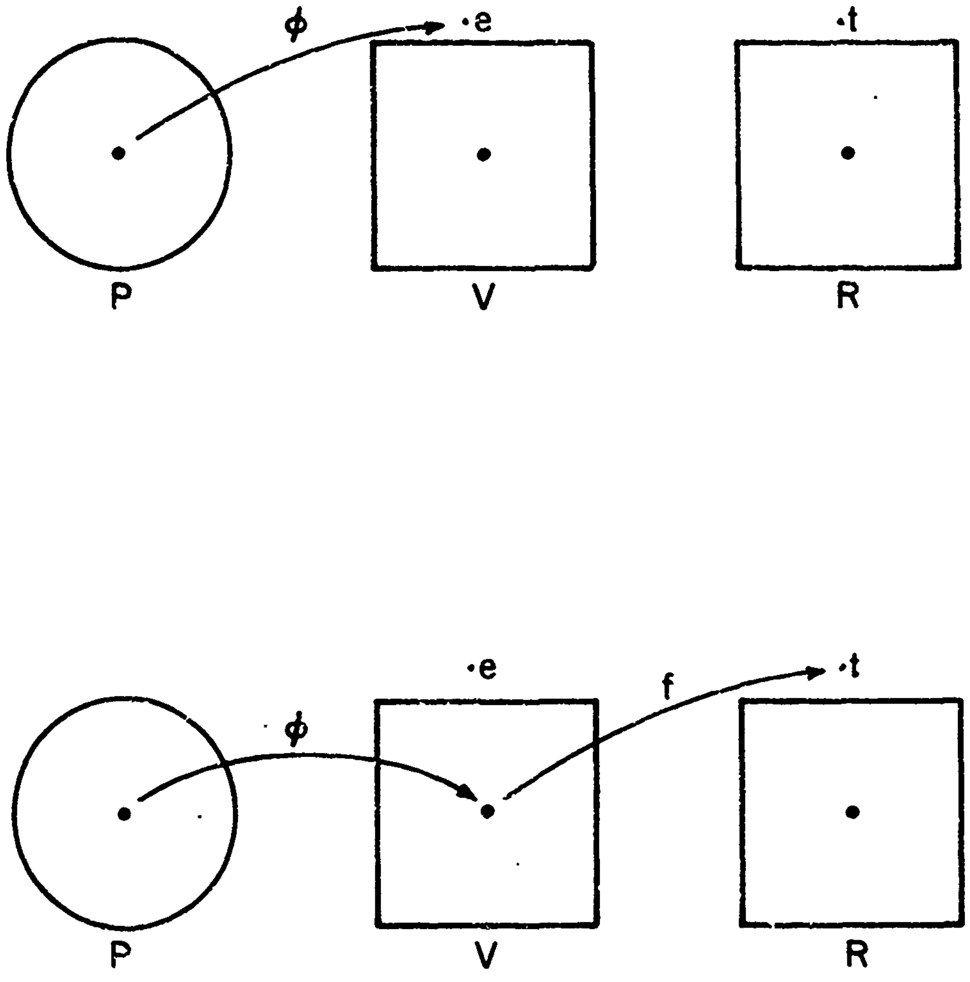
We further distinguish between f and ϕ by contrasting the notions of level and layer in IV generation virtual computer systems. As previously stated f establishes a level relationship between a virtual machine and its corresponding real machine. Thus, f exists as an interlevel map. On the other hand, ϕ is an intra-level map, mapping process names into a particular level of resource names. One characteristic of the ϕ map in IV generation systems will be layering. Thus a IV generation VCS will be multi-layer per level. [See also Section 2.1.]

In the event of a IV generation VCS process name exception, control should be given to a more privileged layer within the same level. A virtual name fault, however, should cause a fault to a process in a lower level number machine. See Figure 4-6.

Another contrast between f and ϕ concerns their visibility by software procedures. Since ϕ is an intralevel map, the inner layer privileged software of that level will likely manipulate ϕ. However, f is an interlevel map. There should be *no way* in which software at level n+1 (regardless of layer) can access the f which maps level n+1 into level n. Thus, the f-map is *invisible* to all software at level n+1

Type II Virtual Machines Revisited

As discussed in "Virtual Machines: Semantics and Examples" [53], and Chapters 2 and 3, a Type II (extended machine host) virtual machine organization is one in which the VMM runs, not on a bare machine as the supervisor, but rather on an extended host under the host supervisor. Depending upon the uses of virtual machines, whether or not there exists one preferred, favored operating system, and whether that operating system has certain capabilities to permit its construction, a Type II virtual machine system can be a useful and possibly easy to implement artifact.



PROCESS EXCEPTION AND VM FAULT
FIGURE 4-6

There is a very similar, corresponding concept to Type II systems in IV generation virtual machines. In a Type II IV generation system, there is a slightly different VMmap f' which maps virtual resource names into corresponding process names for some process running on the real machine. Thus, for virtual names V, as before, and for process names Pr, for a process running on the real machine, we define, for each moment of time, a new function

$$f': V \longrightarrow Pr \cup \{t\}$$

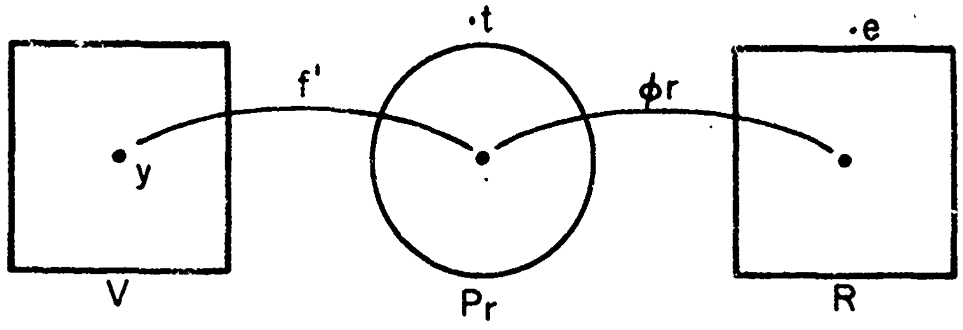
such that if $y \in V$ and $z \in Pr$ then

$$f'(y) = \begin{cases} z & \text{if } z \text{ is the real P-name for V-name } y \\ t & \text{if } y \text{ does not have a corresponding P-name} \end{cases}$$

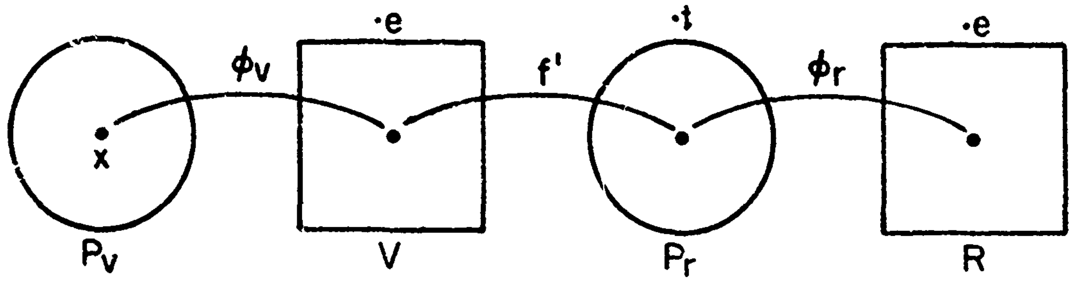
The value $f'(y)=t$ causes a VM fault to occur to some fault handling procedure in the process Pr (or some other process cooperating with Pr).

Since the process Pr is defined on the real machine via (its system base) the function ϕr, a P-name generated by the VMmap is mapped into the corresponding R-name. Thus, we have, for V-name y, the R-name $\phi r(f'(y))$. This mapping is illustrated in Figure 4-7.

Now for any process Pv running on the virtual machine there is defined (via the virtual system base) the function ϕv, which maps the set of P-names, Pv, into their corresponding V-names, V. Thus, the action of running the process Pv causes a P-name, x, to be mapped into the R-name $\phi r(f'(\phi v(x)))$. This mapping is illustrated in Figure 4-8.



TYPE II VMmap
FIGURE 4-7



TYPE II VIRTUAL MACHINE
FIGURE 4-8

Observe that in a Type II system, there exists a VM map f' for each process which creates a virtual machine (or machines) at a higher level number. By invoking the layering of the real machine we can control the ability of processes to create virtual machines and if desired can legislate that there be only one.

Section 4.6 [Example 8] provides a detailed example of a Type II IV generation VCS. However, a simple example, here, may show the intent of this construction. Let f'(x) = 3|x. Then the Type II VMmap's memory component maps the virtual machine's virtual memory into one segment, say 3, of the process. This permits the normal operating system to run as the host on the real machine and perform resource allocation for the virtual machine (on a segment basis).

Recursion

As discussed in "Virtualizeable Architectures" [51] and Chapter 2, recursion for virtual systems is more than a matter of conceptual elegance or a consideration of logical closure. Indeed, it is a capability of considerable practical interest. In its simplest form, the notion of recursion for virtual machines is that, although it makes sense to run standard processes on the virtual machine, in order to test out the VMM software on a VM it is also necessary to be able to run at least a level 2 virtual machine. The real machine is level 0.

In the discussion which follows, we use a Dewey-decimal tree naming convention in which a virtual machine at level n has n syllables in its name [51]. This tree-name is used as a subscript for both the virtual resource space, e.g. V1.1, and corresponding VMmap, e.g. f1.1. Where several different processes are running on virtual systems, the same tree-naming scheme is used, less precisely, to represent the process map on the corresponding virtual machine, e.g. $\phi 1.1$. In this discussion, all f's are of the same form, and all ϕ's are of the same form. The subscripts are used to indicate different entities.

In a Type I IV Generation VCS, if

$f1$: V1 ---> R

$f1.1$: V1.1---> V1

ϕ: P ---> V1.1

then the action of running the process P causes the P-name x to be mapped into f1(f1.1(ϕ(x))) or f1 o f1.1 o ϕ (x). See Figure 4-9.

In this function f1 o f1.1 o ϕ, we identify three possible faults or exceptions:

(1) The process exception to an inner layer procedure of level 2, i.e. ϕ(x)=e,

(2) The level 2 resource (virtual machine) fault to an inner layer procedure of level 1, i.e. f1.1 o ϕ(x)=t.

(3) The level 1 resource (virtual machine) fault to an inner layer procedure of level 0 (the real machine), i.e. f1 o f1.1 o ϕ(x)=t.

For the general case of level n recursion, we have P-name x being mapped into

$$f1 \circ f1.1 \circ \ldots \circ f1. \ldots .1 \circ \phi\ (x)$$

Thus, for a Type I VM, regardless of the level of recursion, there is only one application of the map ϕ followed by n applications of an f map. This is an important result that comes out of the formalism of distinguishing the f and ϕ maps. An obvious result to be discussed later is that in a system with a complicated ϕ but with a simple f, n-level recursion may be easy and inexpensive to implement.

By analogy, we define Type II VM recursion. See Figure 4-10. If

ϕ : Pr --->R

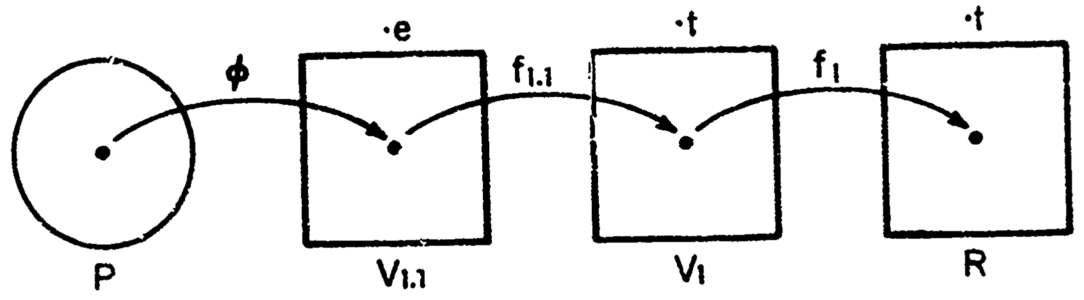
f'1: V1 ---> Pr

ϕ1: P1 ---> V1

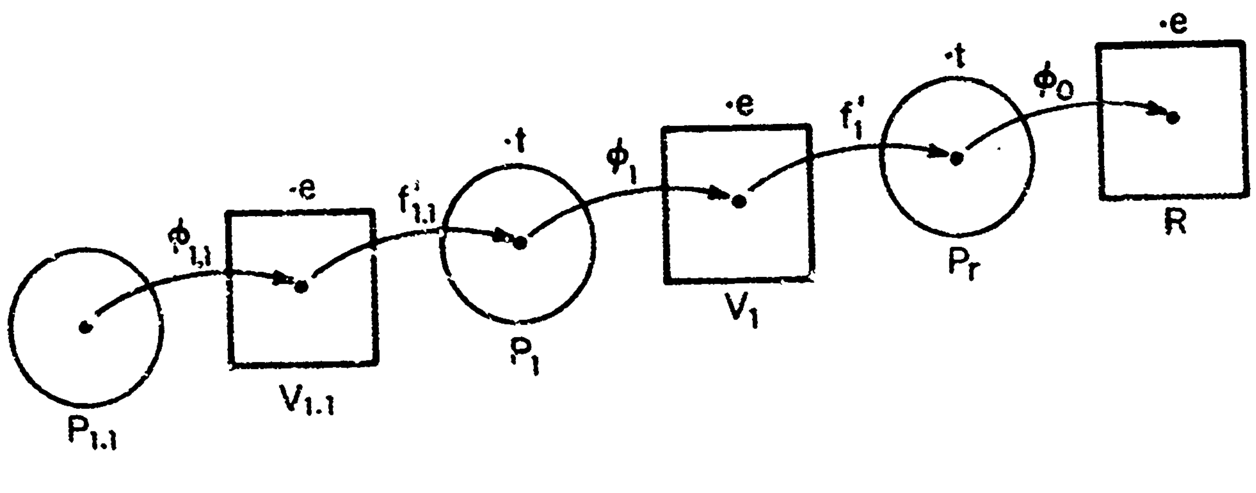
f'1.1: V1.1 ---> P1

ϕ1.1: P1.1 ---> V1.1

Then the action of running the process P1.1 causes the P-name x to be mapped into $\phi \circ f'1 \circ \phi 1 \circ f'1.1 \circ \phi 1.1\ (x)$.



TYPE I RECURSION
FIGURE 4-9



TYPE II RECURSION
FIGURE 4-10

4.3 UNSUITABILITY OF SOFTWARE f

Before dismissing the possibility, it will be necessary to scrutinize IV generation systems to see if a software implementation of the VMmap f, is viable. We will see if the techniques of the III generation VCS can be applied to the IV generation.

As has been developed in Chapter 3, a III generation VCS is normally constructed by:

(1) Using the available memory mapping hardware to help manage the memory of the virtual machine, i.e. virtual memory.

(2) Mapping the master and slave modes of the virtual machine into the slave mode of the physical host machine and simulating the effect of privileged instructions by the VMM.

(3) Trapping all I/O instructions and mapping virtual device names into real devices via software in the VMM. [We will ignore I/O.]

Three of the architectural characteristics that one might expect to be useful in a software implementation of f on a IV generation system are:

(1) Segmented address space

(2) Rings

(3) Emulation.

One might expect the address mapping to be useful in establishing the virtual memory while the ring protection system might provide the layered access control needed to isolate the VMM from its subject VM. As a last resort, one might expect to fall back on the extensive emulation facilities of IV generation systems [5,109].

Unfortunately, as suggested in Section 4.2, none of these features helps the implementation of a VCS. Indeed, they make virtualization potentially more difficult since they will have to be virtualized as well. [This difficulty is similar to the problem noted by Cheatham [25] regarding the self-definition of particularly rich programming languages.] The problem results because segmentation, rings, and emulation are visible, i.e. they are part of the interior decor, and are represented by constructs in the system base which may be seen and manipulated by ("privileged" ring 0) software procedures. We can illustrate this difficulty with a look at rings.

Scrutiny of Rings

A software implementation of the VMmap f on a IV generation VCS requires (1) maintenance of a virtual system base by the virtual machine monitor, and (2) the control of access to the real system base by instructions of the virtual machine [51]. By analogy with the III generation "software VMmap", we prevent access to the real system base by not running ring 0 of the virtual machine as ring 0 of the real machine. Into what ring can ring 0 be mapped?

Two of the properties of rings which must be preserved in any mapping are

(1) monotonicity

(2) finiteness.

That is, the rings define an ordering of privilege: ring i is more privileged than ring j if i<j [57,106]. Furthermore, there is a finite, relatively small number of rings in the system's universe, e.g. Multics has eight [35,106].

Therefore, two ring mapping schemes are presented in Figure 4-11. The first mapping scheme,

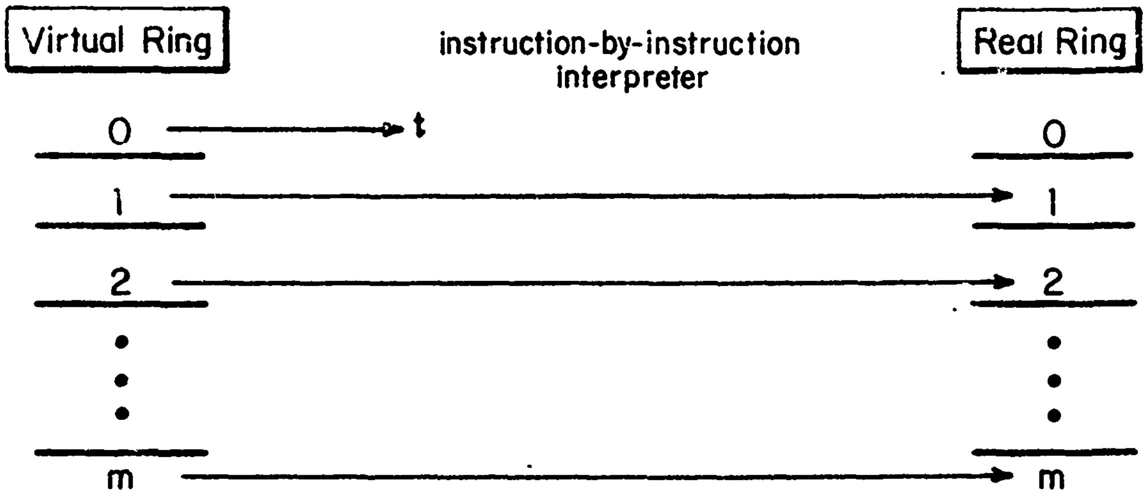
$$f(x) = \begin{cases} x & \text{if } 0<x=<m \\ t & \text{if } x=0 \end{cases}$$

maps rings identically except in the most privileged ring. All ring 0 instructions are simulated on an instruction by instruction basis. An attempt by an outer ring to call a ring 0 procedure must cause a process exception which is understood as a fault to the VMM. This construction is the IV generation equivalent of the hybrid virtual machine, HVM. [See Sections 2.1 and 3.3.] The empirical requirements are similar, e.g. attempted calls to ring 0 can be made to fault. The performance of the IV generation HVM is significantly degraded because of the likely frequent software interpretation of all ring 0 instructions.

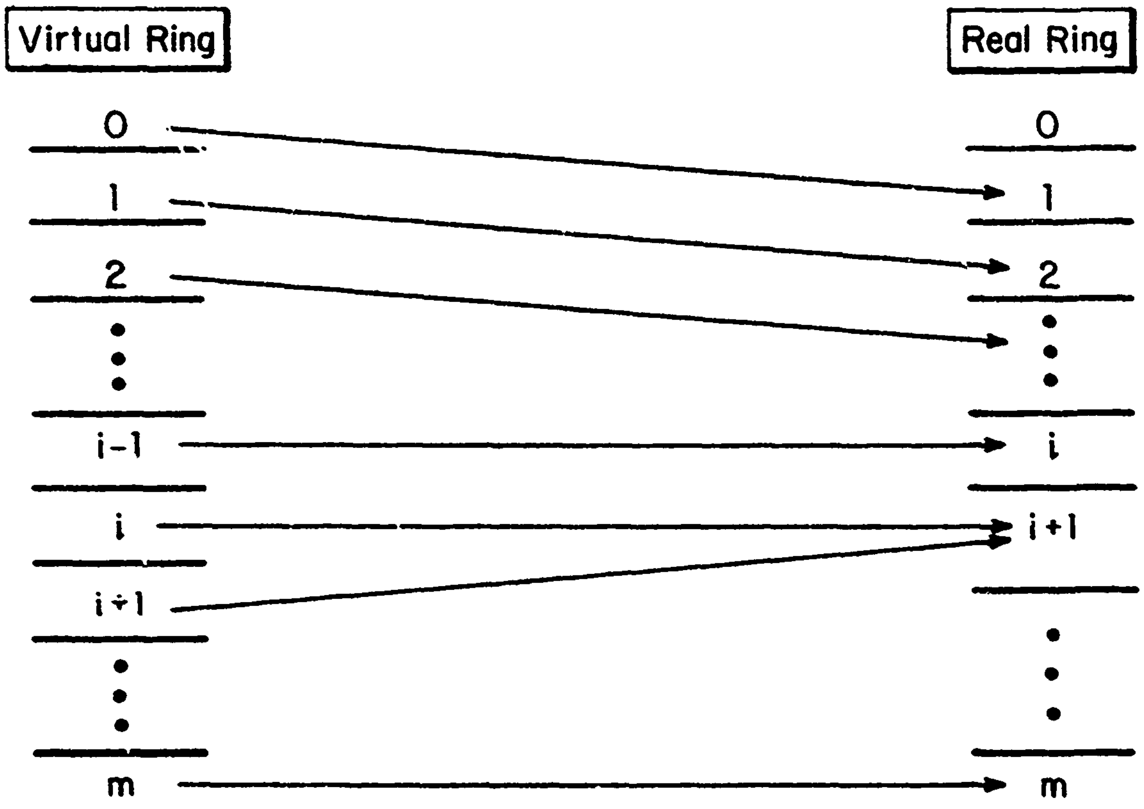
The second mapping scheme,

$$f(x) = \begin{cases} x+1 & \text{if } 0=<x=<i \quad \text{for some } i=<m \\ x & \text{if } i<x=<m \end{cases}$$

maps ring 0 into ring 1. To preserve monotonicity of rings, ring 1 must be mapped into ring 2, etc. However, because of the finiteness property, it is necessary to map a (particular) pair



Ring mapping scheme 1: $f(x)=\begin{cases} x & \text{if } 0 < x \le m \\ t & \text{if } x = 0 \end{cases}$



Ring mapping scheme 2: $f(x)=\begin{cases} x+1 & \text{if } 0 \le x \le i \\ x & \text{if } i < x \le m \end{cases}$

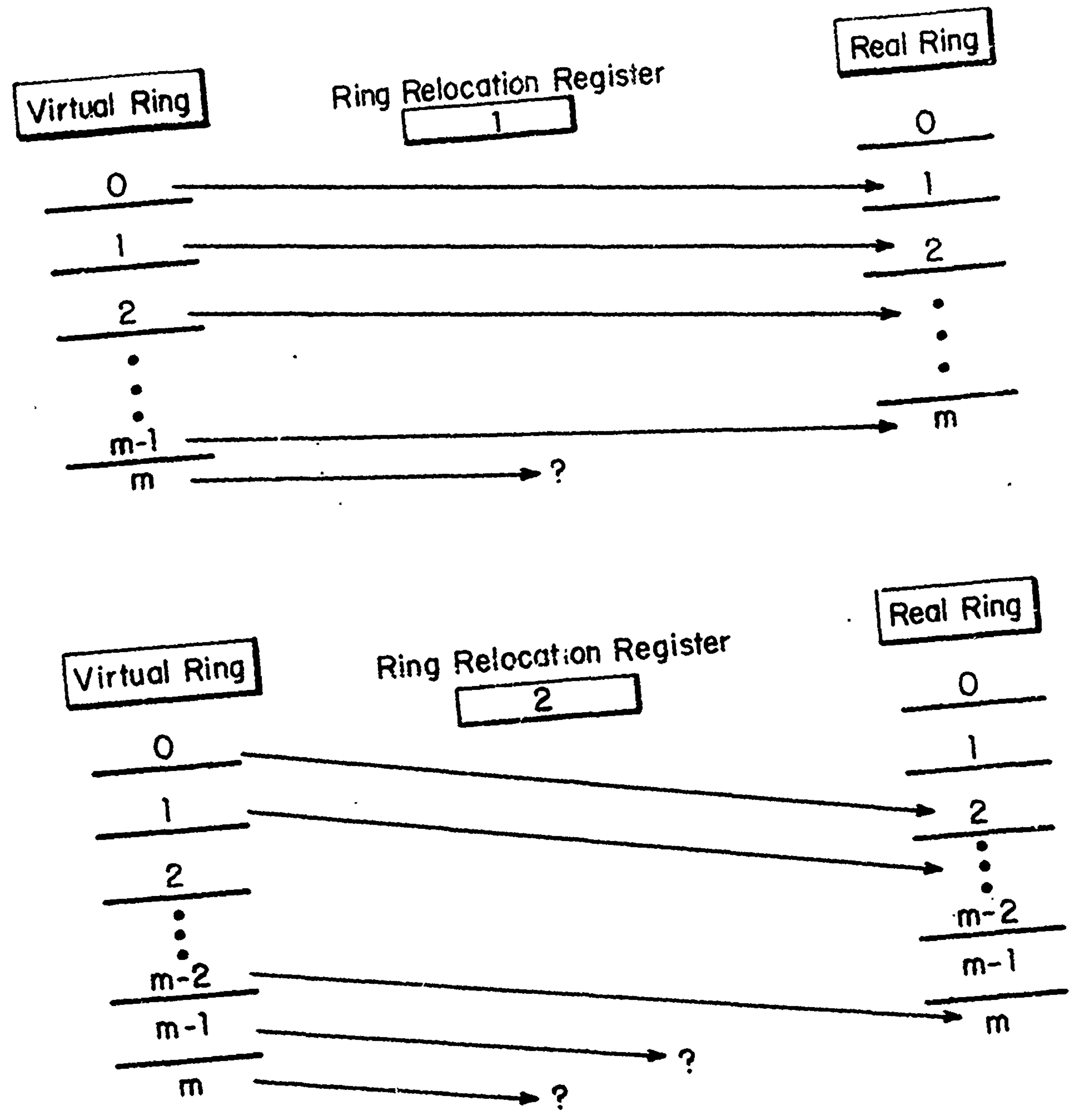
RING MAPPING SCHEMES
FIGURE 4-11

of adjacent distinct rings of the virtual machine to a single ring of the real machine. In a III generation computer system, if master mode is understood as ring 0 and slave mode as ring 1, the processor mode component of the software VMmap for III generation systems [as in Chapter 3] is just the second mapping scheme, f(x), with m=1 and i=2. Analagously with III generation empirical requirements, it is necessary to prevent the processor from determining in what physical ring execution is taking place. This is rather difficult because of the potential visibility of the ring number in the instruction counter ('ring register field') or in stacks, for calls across a ring boundary [106].

Another problem which might develop concerns the absolute interpretation of physical ring number for certain side effects. For example, one proposal [106] has the hardware identify ring numbers with the segment number of the stack segment for that ring, or ring numbers, i.e. 0-1, 2-5, 6-7, with different access properties to gates.

One simple hardware proposal that might be made to avoid some pitfalls of the second mapping scheme is a ring relocation register. See Figure 4-12. The ring relocation register could be used to linearly displace ring numbers and, thus, prevent ring 0 of the virtual machine from having the same privilege as ring 0 of the real machine. However, problems still exist concerning the finiteness property. What ring does ring m get mapped into? In recursion, what ring does ring m-1 get mapped into, etc.?

Thus, the mapping of ring numbers via software or with hardware assistance does not necessarily help in the



EXAMPLE OF RING RELOCATION REGISTER
FIGURE 4-12

implementation of IV generation virtual machines. As we shall see, the software mapping technique is unnatural and unnecessary given a proper understanding of the model of Section 4.2. Fourth generation systems formalize the process map in firmware. Thus, in introducing virtual machines into the IV generation, it is natural to consider how firmware may be used to formalize the VMmap, as well.

4.4 THE VENICE PAPER

In "Virtualizeable Architectures" [51], called the Venice Paper or Venice Proposal, abbreviated as VP, the authors propose an interim solution to the problem of virtualization of IV generation computer systems. The VP observes that because of the high frequency of references to the system base, its maintenance as pure software construct is likely to be doomed to failure. [See Section 4.3 above.] Thus, the solution proposed by the VP is that hardware-firmware be provided to assist virtualization. Each virtual machine will maintain its own system base, visible to itself. However, when the virtual machine is actually dispatched it will be running with a special "translated system base" maintained for each virtual machine by the VMM. Thus, any attempt by software running on the virtual machine to read its system base will be directed to the virtual system base and will go unintercepted. However, any attempt by the virtual machine to write its system base faults to the VMM which can make the corresponding change to the translated system base before redispatching the virtual machine.

Thus, the Venice Proposal provides a major improvement over previous virtual system designs in permitting read access to the system base to proceed without interruption. Only on attempted write access to the system base will the familiar fault-simulate-redispatch sequence be required.

Furthermore, the VP provides a virtualization design that permits a limited two level recursion of virtual machines to operate with hardware dispatching of the appropriate parent virtual machine on a write-access fault to an inferior virtual machine system base. The performance of the second level virtual machine could be comparable to the one level case.

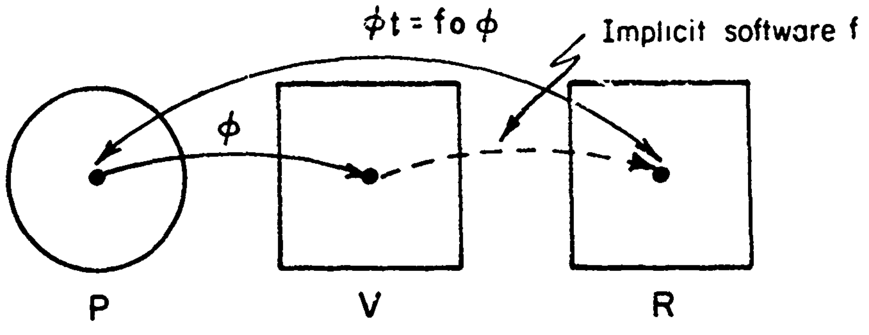
We can interpret the VP in terms of the model developed in Section 4.2. Let P,V,R, ϕ, and f be as before. Then the VP maintains, for a virtual machine, a translated system base,

$$\phi t = f \circ \phi$$

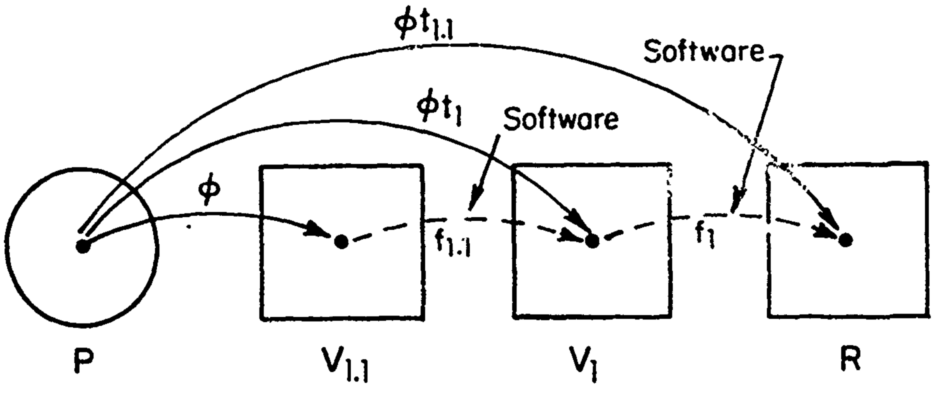
and this is, in fact, the process map that is active when a process of the VM is dispatched. See Figure 4-13. The VMmap f is strictly a software map and never appears in firmware accessed tables; ϕt is accessed by the firmware. The VMM statically composes f with ϕ to produce ϕt. Thus, $f \circ \phi$ is "calculated" in advance of the process being dispatched. Consequently a modification to ϕ cannot be reflected in a corresponding dynamic modification to ϕt. To maintain integrity, a modification to ϕ must cause an immediate VM-fault to the VMM, which calculates a new value for ϕt and redispatches the VM.

Note that this analysis (as with the discussion of Section 4.3) implies that there must be an easy way for the hardware to tell that an attempt is being made to modify the system base. A special instruction might be reserved. [See Section 3.4.] Otherwise, all addresses in a physically "overlayed" segment structure might have to be checked, significantly degrading performance.

A detailed illustration of actual instruction executions in the VP is presented in Section 4.6.



VENICE PROPOSAL
FIGURE 4-13



VENICE PROPOSAL WITH RECURSION
FIGURE 4-14

In the recursive case, tne VP's VMM statically calculates, via software, the translated composed map

$$\phi t1.1 = f1 \circ \phi t1 = f1 \circ f1.1 \circ \phi$$

and this map is, in fact, active when a process of the VM is dispatched. See Figure 4-14. Again, attempted write access by a process to ϕ must VM fault to allow the change to be reflected, via software, in $\phi t1.1$.

Summary

The VP proposes an implementation of a IV Generation VCS. Unlike the purely software techniques discussed in Section 4.3, the VP does not require the "compression" of privileged layers of the real machine to less privileged layers of the host. Consequently the VP should produce efficient IV generation virtual machines. Furthermore, the VP should be feasible in many cases in which pure software techniques are impossible to use. However, the VP suffers from the following disadvantages:

(1) Performance degradation due to faults on system base write instructions

(2) Storage wasted due to translated system base. This could be particularly significant in the recursion example.

(3) Complexity and software needed, in the VMM, for (1) and (2).

In the rest of this chapter, we shall consider a proposed design which eliminates these weaknesses.

4.5 THE HARDWARE VIRTUALIZER

The Hardware Virtualizer (HV) is a hardware-firmware implementation of the VMmap f (for a IV generation VCS). In this section, we shall develop the arguments in favor of an HV, show how an HV can be constructed, show that the construction is feasible in terms of IV generation hardware (realization) components, and indicate that satisfactory performance will occur. As in Section 4.1-4.4 above, because the IV generation treatment of I/O is so similar to III, we do not incorporate I/O into the HV. Similarly, because of the intrinsic problems with time in a VCS we do not discuss time. These difficulties are considered in Section 4.7.

Argument for HV

It is our claim that software construction of the VMmap f is unnatural and unnecessary given a proper understanding of virtualization as developed in the thesis [especially Section 4.2]. In numerous other areas of computer systems research as basic and underlying principles have been developed or discovered, ad hoc and often cumbersome software constructs have been replaced by elegant hardware (-firmware) mechanisms. One can turn for confirmation to management of multi-level memory [28,38,39] or protection rings [57,106]. The firmware implementation of the process model in IV generation systems [51,79,87] is a tour-de-force, including the two previous examples within it as special subexamples.

We claim that the hardware virtualizer HV brings about a simplification of the virtualization mechanism. The HV simplifies and eliminates most of the VMM software, reduces the main storage requirements of the VMM, and produces satisfactory operating performance. Furthermore, it permits virtualization to take place where, using previous software techniques, it ordinarily would not be possible.

HV Construction

Our design of a hardware virtualizer follows directly from the model of Section 4.2. The maps f and ϕ are distinct in a IV generation VCS. Therefore, we implement them as completely different constructs, capable of being referenced and manipulated independently. Execution by the virtual machine causes the maps, f and ϕ, to be composed dynamically to form $f \circ \phi$ at execution time.

The construction of an HV must address the following points:

(1) The database to store f

(2) A mechanism to invoke f

(3) The action on a VM-fault

In the discussion which follows, we shall develop the basis for an HV design somewhat independently of the particular form of the VMmap f which is being implemented. Although we shall refer to certain particular f structures, such as the R-B or paging form of memory map, the actual detailed examples are postponed until

Section 4.6. [The discussion which follows assumes the IV generation process map ϕ presented in Section 4.1. As we note in Section 4.8, suitable simplification and interpretation of the ϕ-map allows us to extend this construction to second and third generation architectures as well.]

Database to Represent f

The VMM must create and maintain a database which represents the VMmap f for the next level virtual machine(s). This database must be stored such that it is invisible to the virtual machine, including its most privileged software. Let us assume that for economic reasons [51] the database must be stored in main memory. Then the VMmap itself must be such that f is not in the (virtual) memory of the virtual machine.

The VM database, logically should be an addition to the system base. A pointer at a fixed location from the ROOT points to a Virtual Computer System Table (VCSTAB). Each entry in the VCSTAB points to a Virtual Computer System Control Block (VCSCB) or colloquially, VMCB. Thus, the i-th entry points to the control block for virtual computer system i.

The VCSCB provides the implementation of the VMmap f for the virtual machine. It contains the memory map, processor(s) map, and, at least conceptually, the I/O map. In addition, there may be other status and/or accounting data for the virtual machine.

As noted in the VCS model (Section 4.2), the maps ϕ and f are completely different. In particular, the memory components of the maps serve different purposes and so are likely to be of different form. The ϕ map likely will be segmented. Depending upon how the VM's will be used, the f map might be R-B, paged, or multi-level paged. If a limited number of virtual systems are used irregularly for operating system debugging or subsystem transferability, an R-B map might be reasonable. If fictitious memory resources are required, some form of page map may be used. [See performance/cost discussion below.] When we discuss a page map, we, usually, do not illustrate fictitious resources, e.g. demand paging.

The processor component of the VCSCB includes (per processor of the VCS) the saved values of the processor registers (the last time the VM was running). Note that this is different from the per process saved register values kept in the respective PCB's of each process on the VM.

These registers typically might include:

P - current process ID

IC - instruction counter

T - top of stack

STR - status

BR - base registers

GR - general registers

SR - scientific registers (floating point)

[In this discussion, we assume that ROOT is zero for simplicity.]

Additional processor related information kept in the VMCB includes capability information, for the particular virtual processor (not the real processor and not processes running on the virtual processor) indicating particular features and instructions, present or absent. These include, for example, scientific instruction set, VM instruction set (LVMID instruction), etc. Additional information might include a general opcode fault mask or a fictiticus resource mask. The opcode mask can be used to force certain opcodes in the VM to fault to the VMM. [This notion is derived from Section 3.4.] A fictitious resource mask might indicate whether (bounds type) faults are to be directed to the VM or the VMM.

<u>Mechanism to invoke f</u>

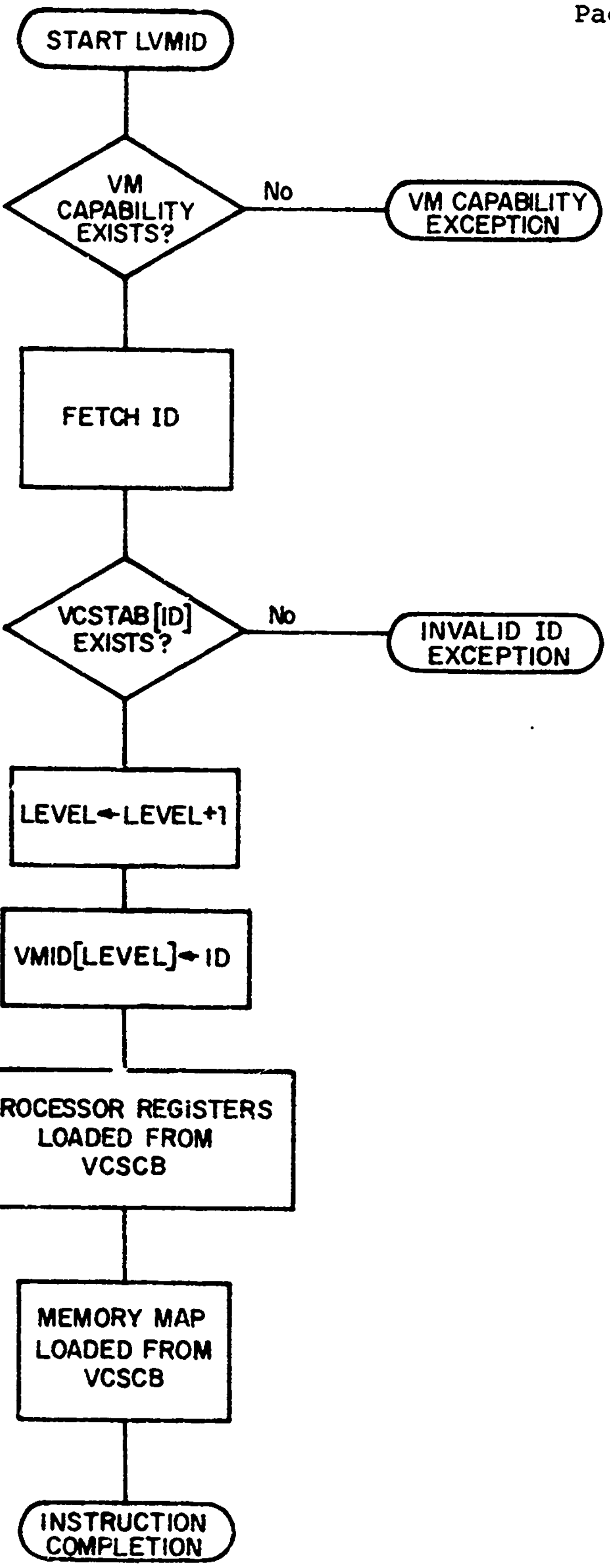
In order to invoke the VMmap f, the HV requires one additional visible register and one instruction for manipulating it. The register is the VMID and contains the number of the VCS in the VCSTAB that is currently executing. The new instruction is LOAD VMID, LVMID. See flowchart in Figure 4-15.

For the hardware virtualizer design to be successful, the VMID register (and the LVMID instruction) must have three crucial properties [51].

(1) The VMID register absolute contents may neither be read nor written by software.

(2) The only modifications possible to the value in the VMID register are accerding a low order ID syllable (as a result of an LVMID instruction) or removing some number of low order ID syllables (as a result of a VM-fault).

(3) The VMID of the real machine is the null



LVMID INSTRUCTION
FIGURE 4-15

identifier.

These properties all follow directly from the desire to model the relative resource contexts provided by invocation of f.

Needless to say, LVMID "loads" (appends) the VMID with the ID specified as the operand of the instruction. If the VM capability does not exist for this machine, a VM capability exception occurs, which is a process exception to a privileged procedure in the same machine. If the VMID is loaded successfully, the firmware checks the appropriate VCSTAB entry for validity. If it is marked as absent or invalid, a VCSTAB absent or invalid exception occurs to a procedure in the same machine.

Otherwise the processor enters virtual mode and (assuming no faulting because of VCSCB values) loads the processor registers (P, IC, BR, etc.) from the VCSCB.

Now the memory component of f is active, say relocation and bounds, R-B. This causes a loading of the R-B into scratchpad registers, say, Rcomp-Bcomp. Thus, every virtual memory name will be transformed to a real name by the R-B map, i.e. value currently in Rcomp-Bcomp. [Of course, the scratchpad registers Rcomp-Bcomp, may be neither read nor written by any software. When the real machine (VMID=null identifier) is running, Rcomp=0 and Bcomp=physical memory size.] In a segmentation system, the result of looking up a segment P-name and obtaining the V-name must be subjected to the R-B map. However, since the segment table is not the root of the system base, the firmware must first locate it from the true ROOT and the process's P-name, i.e.

process-id. This endeavor requires the firmware to make several references to main memory in the course of "walking" the system base list structure. Each of these references to the virtual memory must be mapped to the real memory equivalent by applying R-B, i.e. adding Rcomp-Bcomp.

Eventually, the location of the segment table is established and it is saved in a scratch pad register. Thus, any subsequent memory reference may directly utilize the segment table to obtain a virtual memory location which is then mapped, via R-B, into its real equivalent.

For reasons of performance, the original machine requires an associative memory to store the most recent 'segment number-memory location' pairs. Then, the hardware virtualizer's associator will store 'segment number-real memory location' pairs. On setting a new VMID, the associator can be cleared. Another possible choice, which can be made on the basis of cost/performance, is to have a still larger associator and keep the VMID as a field to be used as part of the search key.

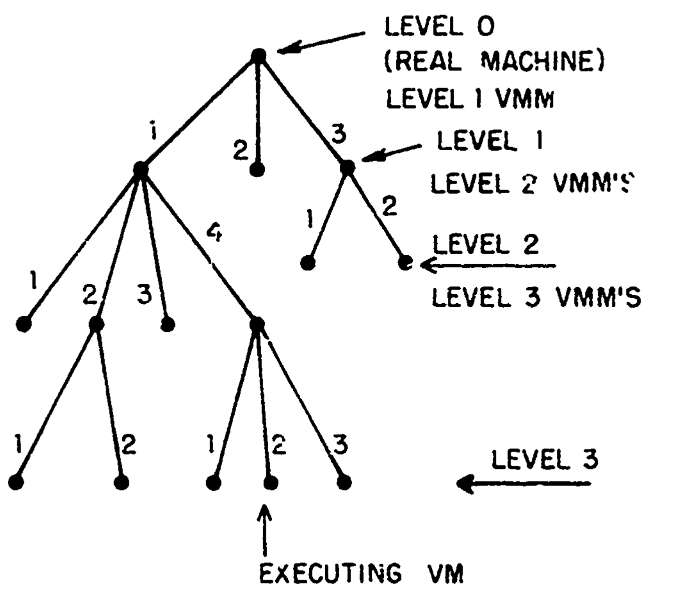
Recursive invocation of virtual machines adds little additional complexity to this implementation. If the VM capability bit in the VMCB is set to one, then the virtual machine has the same (software visible) hardware virtualizer VM features as in a real machine. Therefore, a VMM running on the VM may attempt to create an inferior virtual machine by creating a VCSTAB entry and a VCSCB for the level 2 VM in its system base. It may also dispatch the (level 2) VM by executing an LVMID instruction.

As noted in the Venice Paper [51] and above, recursion requires that the VMID be made a "multi-syllable register". When the real machine is executing, the VMID value is null. When the VMM dispatches a VM, it loads (appends) a value into the VMID, say 1. If that virtual machine is running a VMM which creates and dispatches a VM, say ID 1, then 1 gets appended to the VMID. Thus, the new composite value in the VMID is "1.1" which gives the ID of the currently executing VM. As stated above, this value may not be either read or written by any software of the virtual machine. See the tree structure shown in Figure 4-16.

To handle a small, finite depth of recursion requires the VMID register to be sufficiently "wide" for specification. In practice, one should not expect the VM level to go more than 3 or 4 deep.

As in the non-recursive case, the execution of the LVMID instruction by the level 1 VMM causes its VCSTAE and VCSCB entry to be checked for absence or validity. If the LVMID completes successfully, the machine is now in virtual mode at the next level.

Now the memory component of the new f is active. [In this discussion, a map of the form R-B.] This causes an addition of the new R-B to the scratchpad register Rcomp-Bcomp where the current R-B value "total" is being maintained. However, in addition to the running sum, a collection of scratchpad registers are set aside to hold the R-B values of each level. [As Lauer and Snow [76] observe, it may also be possible for the firmware to store the reverse "trail" in main memory.] This permits



VM TREE STRUCTURE
FIGURE 4-16

restoring the previous composed map by subtracting on a VM-fault. In terms of the VCS model, the running sum of the R-B's, Rcomp-Bcomp, provides a calculation of the composition of the two f maps (in the memory dimension), say f1 o f1.1. Then the composed map is combined with the ϕ-map.

Thus, every virtual memory name will be transformed to a real name by the composite R-B map, Rcomp-Bcomp. This is done, just as in the single level case, with the composite R-B map taking the place of the simple R-B map. This includes walking the level 2 system base, etc.

Just as in the single level case where an associative memory was used to store 'segment number-real memory location' pairs, so in the recursive case, an associator stores 'level 2 segment number-real memory location' pairs. The performance implications of the associator (map composer) design will be discussed below. Section 4.6 provides detailed examples, including illustrations of the associator during HV operation.

Our discussion of the VM invocation has utilized an R-B memory component in the VCSCB. The mechanism described works for other memory maps, as well. [However, we are assuming that there is only one f map structure, common to the entire system.] For example, a paged memory map might be stored in the VCSCB. After the LVMID instruction, map composition in the memory dimension is dynamically performed on a page basis. This implies that the associator for a paged HV is of a slightly different form from the R-B HV. Illustrations are provided in Section 4.6.

VM-fault

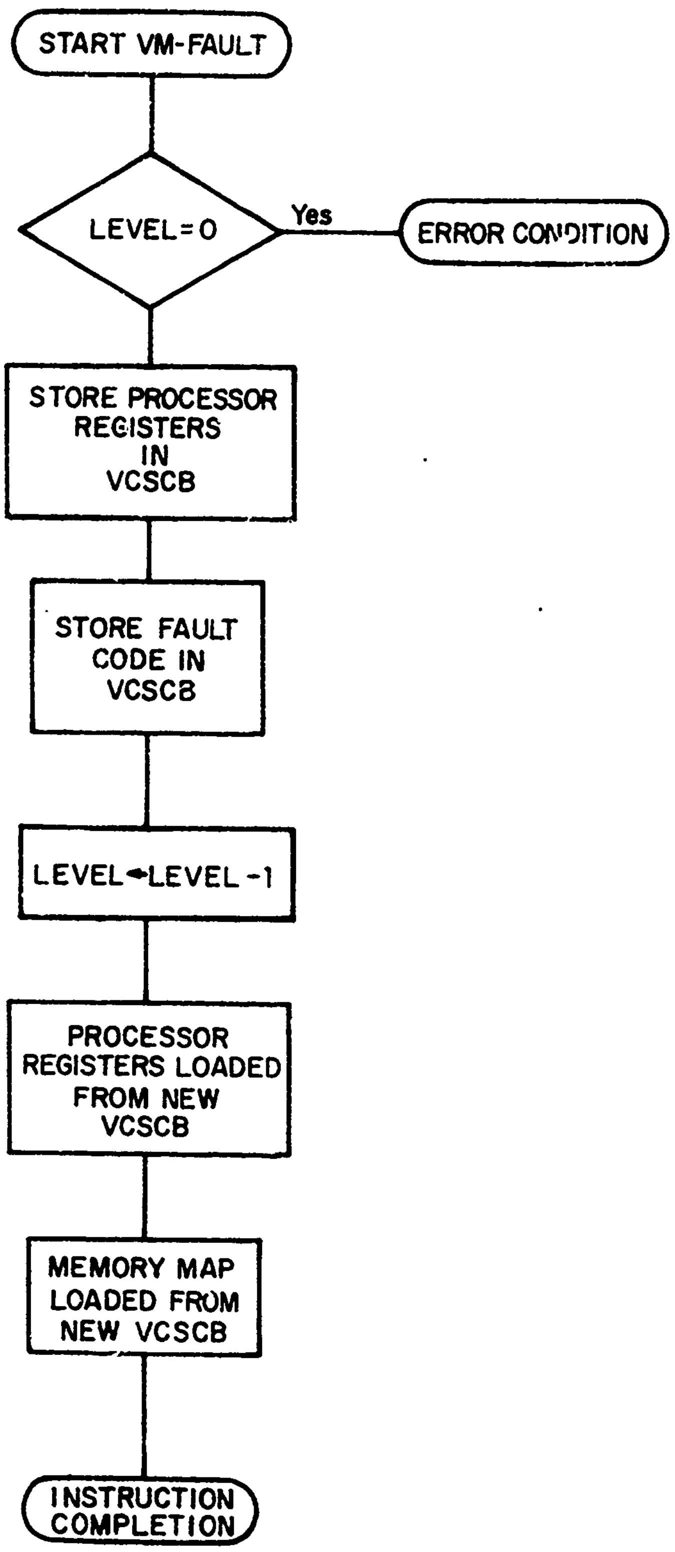
On a VM-fault (VM-level fault), control must pass to the VMM. If the VMM is running on the real machine, control passes to the real machine. If the VM is running at level n, then control passes to the VMM at level n-1. The faulting mechanism (firmware) must inspect the VMID register and discard the low order syllable. The resulting value is the ID of the VMM's machine. See flowchart in Figure 4-17.

The characteristics of the VM-fault can be derived from the empirical work with III generation systems [Chapter 3]. In particular, the VM-fault must be completely invisible to the VM. The VM can have no way of testing or discovering whether a fault has occurred.

On a VM-fault, the processor's registers must be saved in the VCSCB and the associator must be purged. A fault code can be stored in the VCSCB and the VMM is started. After processing the fault, the VMM may return to the VM by issuing an LVMID instruction.

Feasibility of Construction

The HV described above is feasible to construct on a IV generation system. The hierarchical levels of control and map composition facilities add little additional complexity to the process model. Three characteristics of IV generation realizations [19] will be the use of large control stores, elaborate micro-logic, and associative memories. The hardware


START VM-FAULT
LEVEL=0
Yes
ERROR CONDITION
STORE PROCESSOR REGISTERS IN VCSCB
STORE FAULT CODE IN VCSCB
LEVEL←LEVEL-1
PROCESSOR REGISTERS LOADED FROM NEW VCSCB
MEMORY MAP LOADED FROM NEW VCSCB
INSTRUCTION COMPLETION

VM-FAULT

FIGURE 4-17

virtualizer will utilize these same building blocks.

Performance Assumptions and Analysis

The expectation of acceptable performance by processes running on the virtual system depends, to some extent, on the expectation of acceptable performance in the real system. As discussed above, the process map ϕ in IV generation systems will likely involve rather elaborate constructs which must be accessed by the firmware for every instruction execution. For example, in the case of segmentation, it will be necessary for the firmware to obtain the resource location names (from P-names) via ϕ for every instruction. This will typically require:

(1) The most recently referenced segments to have P-name, R-name pairs stored in some small finite number of associative registers, and

(2) The recently referenced segments to be maintained in main memory by the software.

Needless to say, any architecture which incorporates paging (as well as segmentation) in the ϕ map will certainly be implemented this way. [As we shall see below, paging properly belongs in the f-map instead.]

That a IV generation architecture (with segmentation, etc.) should perform acceptably depends significantly on the principle of locality. From the initial notion of "program locality" as noted by Dennis and Denning, Madnick [60] has generalized and identified two specific aspects of locality that are used.

(1) Temporal locality

If the logical addresses <a1, a2, ...> are referenced during the time interval t-T to t, there is a high probability that these same logical addresses will be referenced during the time interval t to t+T.

This behavior can be rationalized by program constructs such as loops, frequently used variables, and frequently used subroutines [80].

(2) Spatial locality

If the logical address a is referenced at time t, there is a high probability that a logical address in the range a-A to a+A will be referenced at time t+1.

This behavior can be rationalized by program constructs such as: sequential instruction sequencing, and linear data structures (e.g. arrays) [80].

In IV generation systems, because of the complex ♦ and the cost required to "start up" the map, it is likely that the (software) scheduler and (firmware) dispatcher will enforce an additional locality:

(3) Process locality

If the process identifier of the process executing at time t is P, then there is a high probability that it will be P at time t+1.

Virtual machines and the hardware virtualizer add a new notion.

(4) Virtual machine locality

If the VMID of the currently executing VM at time t is x1.x2. ... .xn-1,xn, then there is a high probability that the VMID will be x1.x2. ... .xn-1.xn at time t+1. Furthermore, the other values that it can take are x1.x2. ... .xn-1 or x1.x2. ... .xn-1.xn.xn+1. The first value occurs on a VM-fault, the second on recursion. On multi-level faults, other values are possible.

VM locality implies that, with proper implementation, one level or multi-level recursive virtual machines need not have significantly different performance from real machines. Another way of phrasing this observation is:

<u>Temporal and spatial locality are name invariant</u>.

Regardless of what a page-size block is called, or how many times it gets renamed (via a VMmap) there is still an intrinsic probability of reference to it by the executing process. Thus, a map composer aided by an associative store should provide comparable performance in the virtual machine to that obtained in the real machine.

We examine the performance of a virtual machine vs. a real machine for the R-B virtualizer and for the paged virtualizer. Performance of the HV can be judged by a direct comparison of instruction execution rates of the VM vs. the real machine. The use of a cache, look-ahead, etc. makes the comparison difficult. However, we can constrast the time required to develop a physical memory address which then gets sent to the memories.

In the R-B virtualizer, as sketched above, let the following parameters represent access times, a - associative memory

M - main memory

s - scratchpad memory

Let pk(R) be the probability that a segment entry will be found in an associative memory of size k, in the real machine. Let pk(V) be the corresponding probability for our virtual machine. Then pk(V)=pk(R). Let us fix k so this probability is just p. Then the average time to develop a physical memory address is approximately

$Ar = a*p + M*(1-p)$ for the real machine

$Av = a*p + (M+s)*(1-p)$ for the virtual machine

Since the scratchpad maintains a cumulative value of R-B, Rcomp-Bcomp, during recursion, Av is constant regardless of the level VM. Reasonable parameter choices are a=50 nsec, M=500 nsec, s=50 nsec, and p=.99. Then Ar=54.5, Av=55, and Ar/Av>.99. This result implies that, for an R-B virtualizer, regardless of the level of virtualization, Av is approximately equal to Ar.

In the paged virtualizer, the associator is of different form from that of the real machine. Whereas the real machine's associator relates segments to physical memory, the associator in the paged virtualizer must relate page-size blocks within a segment to the physical location of the page-size block. Therefore, in order to contrast the performance of real and virtual machines, we let pk be the probability that a segment page-size block entry will be found in an associative memory of size k. For a particular value of pk, k will be larger in the segment-page associator than in the simple segment associator.

The parameters a and M are as above. We assume that for each level VM a scratchpad register holds the base of the page table. Furthermore, we assume that all scratchpad references are overlapped with main memory fetches and so are ignored for this analysis. Then

$$An = a*pk + (n+1)*M*(1-pk)$$

where n is the level of VM. If we set n=0, we obtain the result for the real machine,

$$A0 = a*pk + M*(1-pk)$$

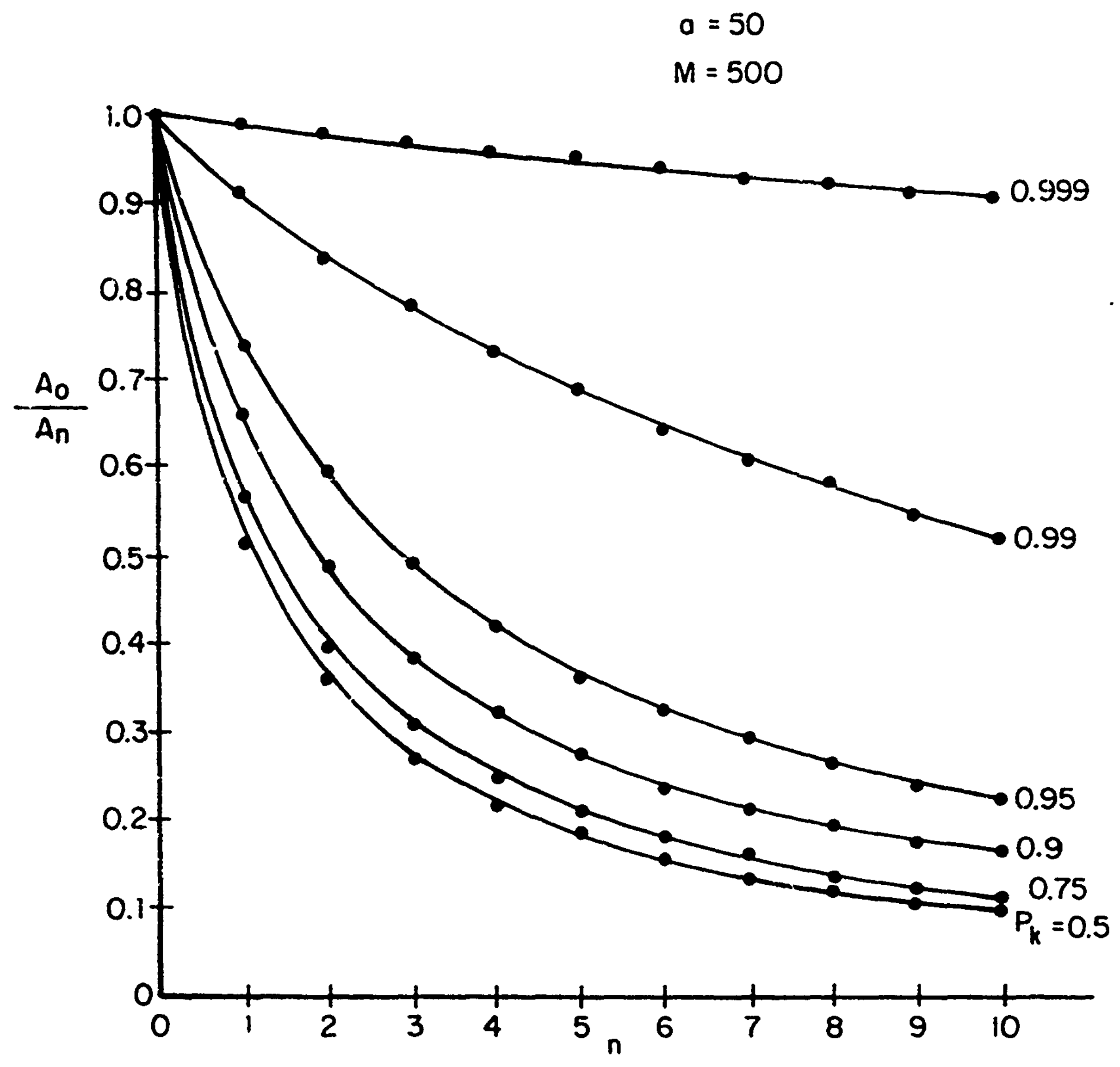
Figure 4-18 plots the ratio A0/An as a function of n for a=50 nsec, M=500 nsec and representative values of pk. In practice n is unlikely to be greater than 3 or 4. Figure 4-19 plots A0/An as a function of pk for n=1,2,3,4.

From the graphs, it can be seen that performance of a paged HV will be respectable, depending upon the environment and application, for a range of values of pk. For n=1,

pk	A0/A1
.900	.66
.950	.74
.990	.92
.995	.95
.999	.99

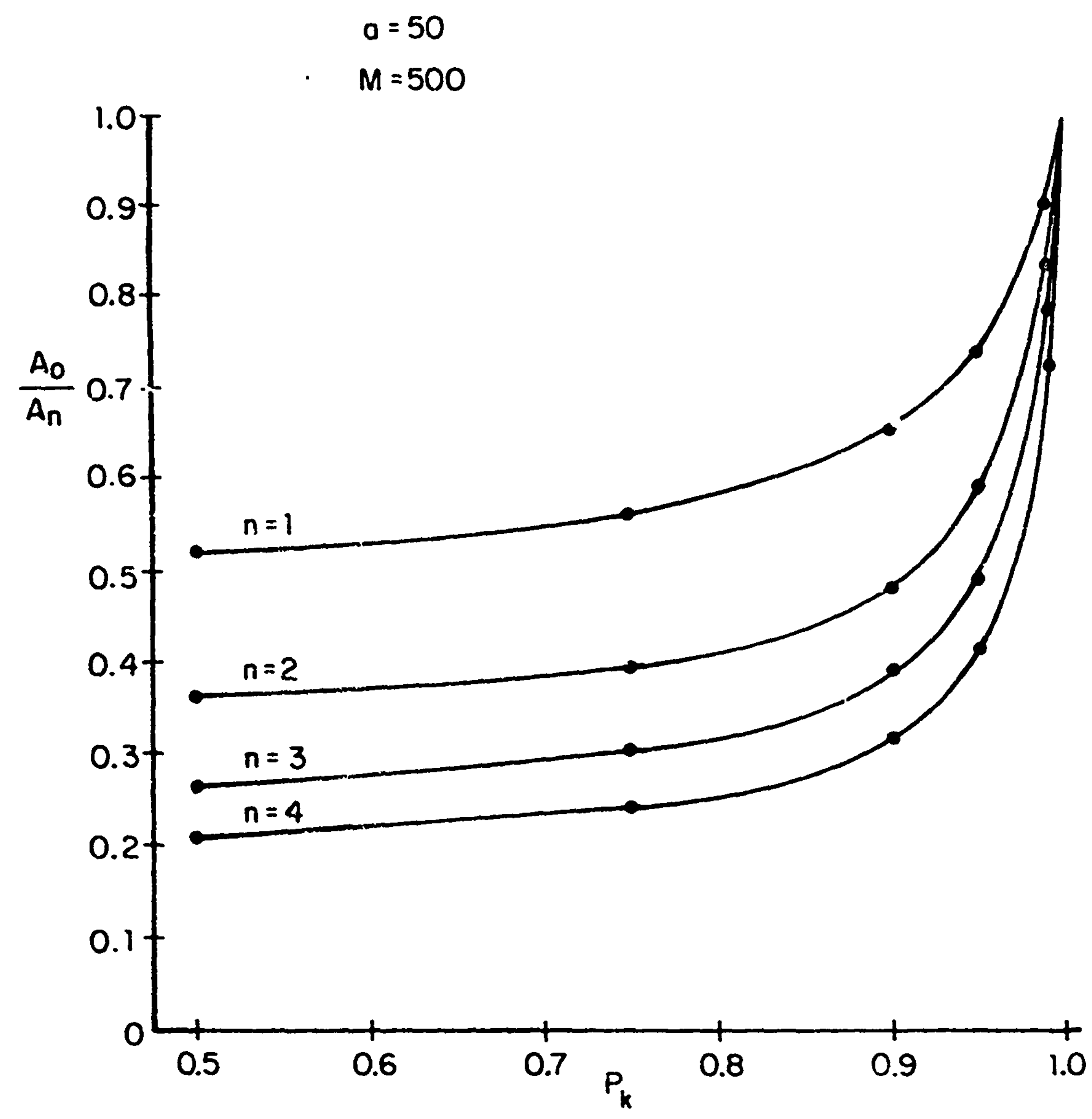
Through choice of k, pk can likely be made to be .99 [105]. Then, the virtual machine runs at 92 percent the speed of the real machine (at least as far as address development goes). This speed might be satisfactory for a normal production environment.

With pk=.99, a level 2 VM runs with A0/A2=.84. This is likely to be satisfactory for debugging the VMM.



HV PERFORMANCE vs. n

FIGURE 4-18



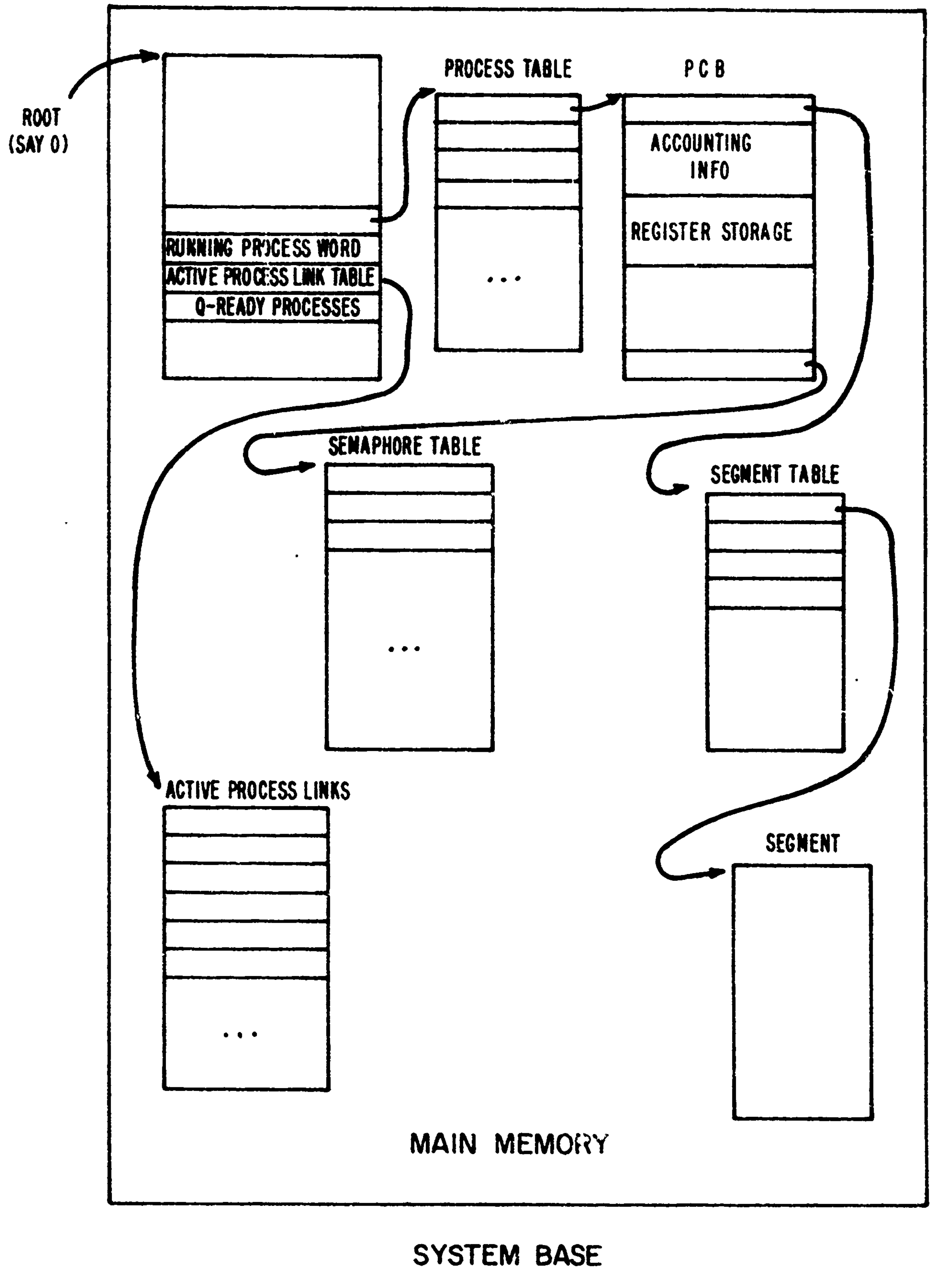
HV PERFORMANCE vs P_k

FIGURE 4-19

4.6 DETAILED ILLUSTRATIONS

In the previous sections, we have indicated the structure and operation of typical IV generation hardware virtualizers. The intent of the present section is to make this operation perfectly clear by illustrating with a number of detailed concrete examples in which we show the execution of actual instructions by the VM. Most of the examples provide the same sequence of instructions being executed by (similar or) identical processes on the virtual machines. This is done so as to provide a basis of comparison between the different VMmaps. The examples generally use stylized instructions; i.e. LOAD, STORE, P, V, SIGNAL--PROCESSOR, and do not worry about accumulators, base registers, etc. Furthermore, the examples do not cover fictitious (memory) resources, I/O, time and other potential special difficulties. The claim is that CPU-memory processes are sufficiently interesting and complex in IV generation systems. If we can run them well in a virtual machine we have accomplished much.

Figure 4-20 shows our stylized typical IV generation system base located in main memory. All pointers illustrated in the figure are absolute storage locations. Associated with each pointer is the size of the table that is being pointed to. The firmware has available to it the base address of the system base, ROOT. The Running Process Word (RPW), Process Table Pointer, etc. are located at fixed, known displacements from the ROOT. Thus, the firmware may locate information about any process by



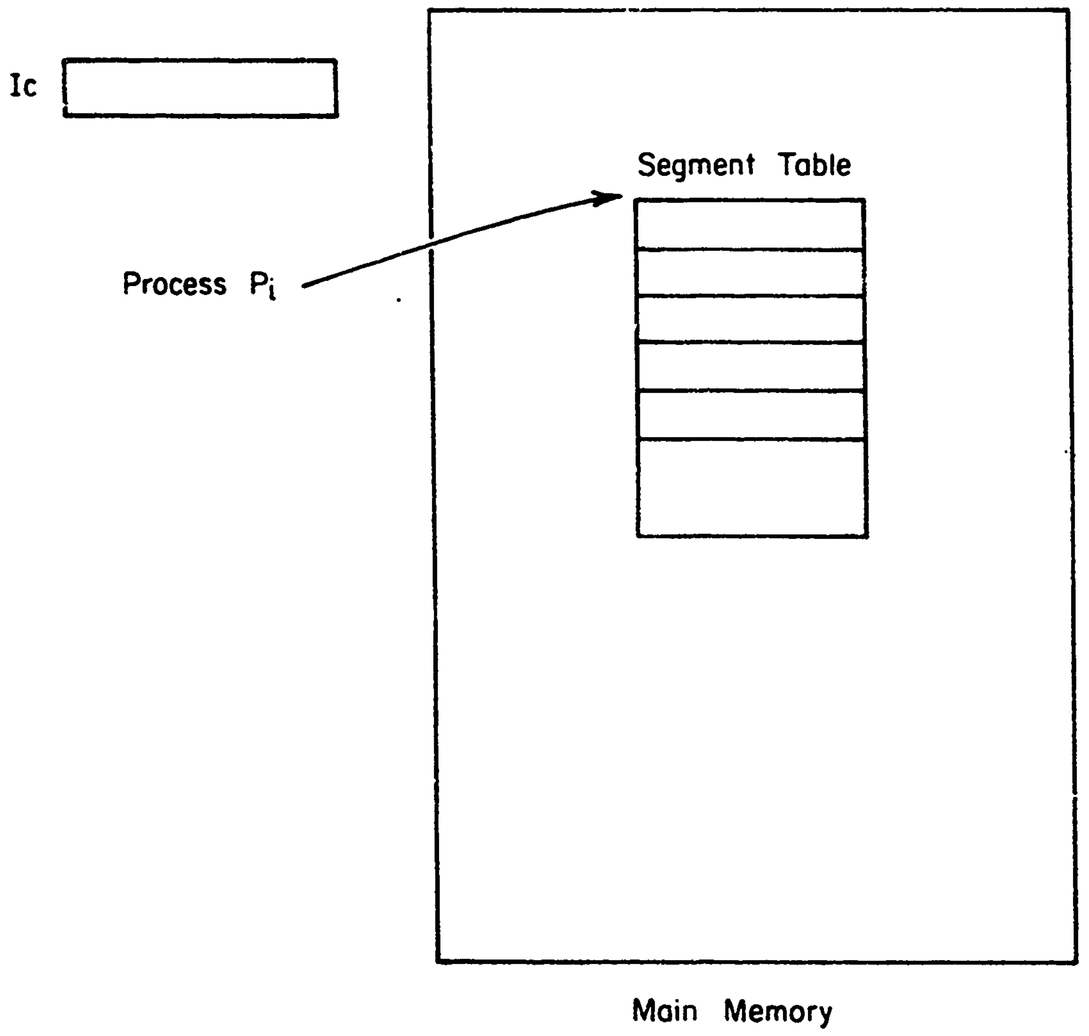
SYSTEM BASE

FIGURE 4-20

starting at the ROOT and tracing through the system base. The ith entry of the Process Table points to the Process Control Block (PCB) of process i. The PCB stores accounting information, CPU register values (for this process), etc. There are also pointers to a semaphore table and a segment table. The semaphore table is discussed, together with the Running Process Word, Active Process Link Table, and G-Ready Process below in the example illustrating P and V operations on semaphores. [See Example 6 below.] Each entry of the segment table is a segment descriptor which is a pointer to that segment (if present in main memory), together with status, and (ring) access information. While we assume a layered ring structure, none of the examples presented below will use it explicitly and so we will represent a segment descriptor by its location and length. The handling of access violations will be very similar to the bounds violations which will be illustrated below.

Rather than reproducing this complete system base in the examples, below, we will usually just show the segment table of the currently running process as in Figure 4-21. This is just the memory component of the process map δ. This, of course, implies that the name of the currently running process, Pi, is stored in the Running Process Word, RPW.

In the examples, we will also indicate the current value of the executing processor's instruction counter, IC, as a segmented address. This is the IC value that, say, has been loaded from the PCB when process Pi was dispatched. Again, we will not indicate the IC's ring information in the examples.

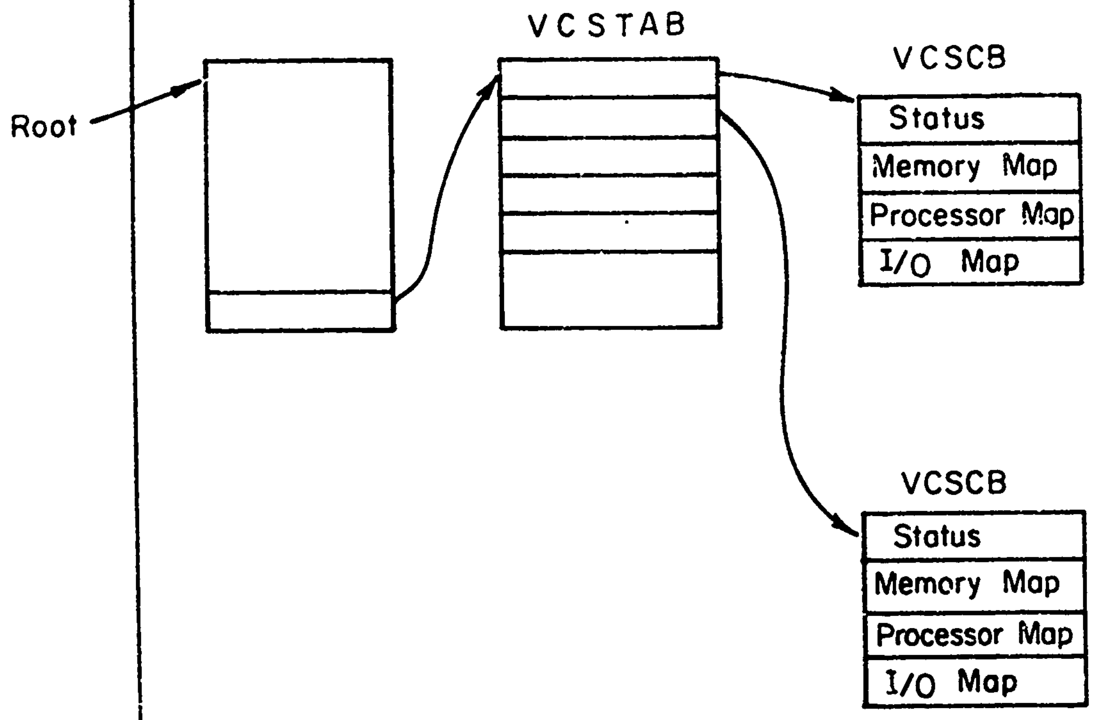


NOTATION FOR SEGMENT TABLE OF RUNNING PROCESS

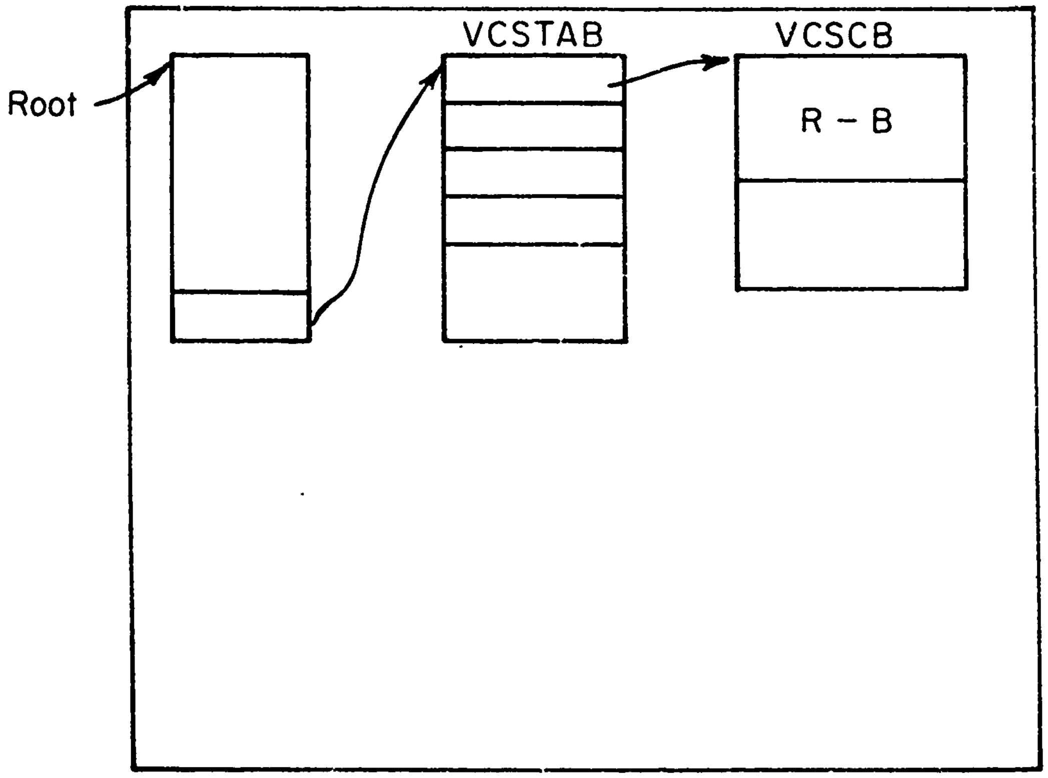
FIGURE 4-21

In Figure 4-22, we show the extension to the system base that we will use for virtual machine support in the Type I VCS examples. All of the structure shown in Figure 4-20 is present here. However, in addition, we now have a pointer at a fixed displacement from the ROOT to the Virtual Computer System Table (VCSTAB). [Note: In general, all pointers have the size of the table they point to stored as a field. Thus, attempted accesses may be checked.] This table has one entry per virtual computer system, which is a pointer to its Virtual Computer System Control Block (VCSCB). The VCSCB has, say, four parts: general status information, the virtual memory map, the virtual processor(s) map, and the virtual I/O map. The examples, below, will not discuss the status information or I/O map. The I/O discussion follows in Section 4.7. Thus, we will normally show just two components of the VCSCB in the drawings. Most examples will illustrate a one processor computer system in which case the processor map reduces, largely, to the storage of register values. [These are, of course, the register values of the processor when it VM-faulted, not of any particular process being run.] Under those conditions, the VCSCB is sometimes called, colloquially, the Virtual Machine Control Block (VMCB).

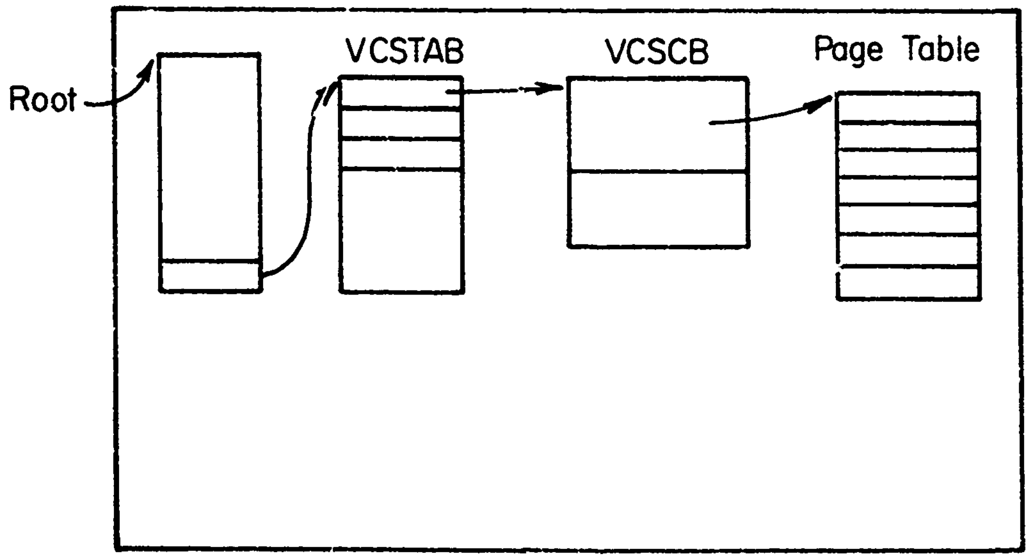
The memory map component of the VCSCB is illustrated for several different memory maps in the examples. In the examples of a conventional relocation and bounds memory map R-B, the values are shown stored in the VCSCB. The values are represented as the relocation value and the length of the contiguous allocation. See Figure 4-23. In the examples of a conventional



THE VCSTAB and VCSCB'S
FIGURE 4-22



NOTATION FOR R-B VCSCB
FIGURE 4-23



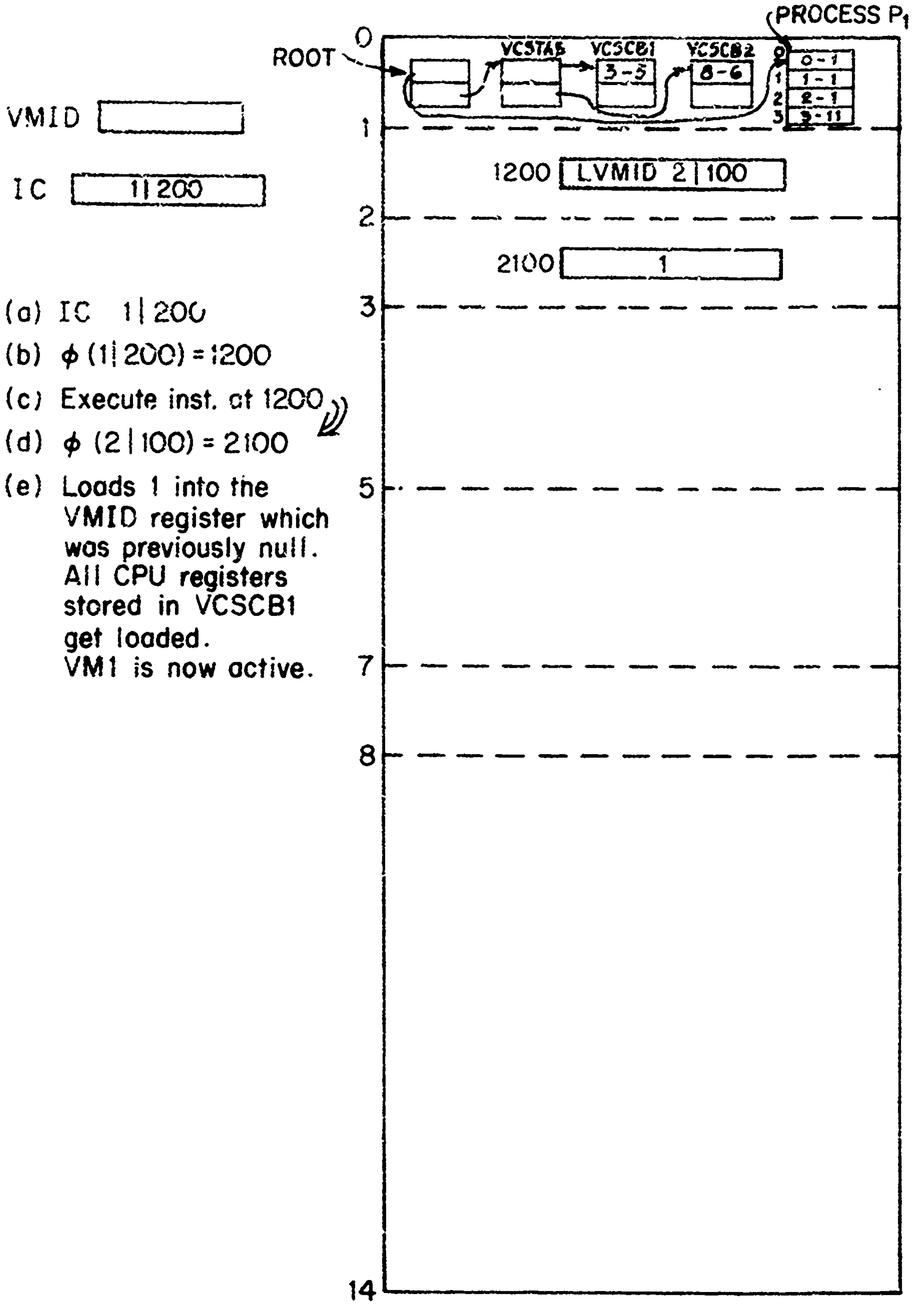
NOTATION FOR PAGED VCSCB
FIGURE 4-24

(fixed size block) paged memory map, the VCSCB contains a pointer to the page table. See Figure 4-24. The page table may contain status information such as presence, usage, etc. These considerations are essentially the same as that discussed for III generation virtual computer systems [see Chapter 3]. In our examples we do not illustrate fictitious memory resources and always show all pages in main memory. [This, of course, is not a requirement.] Thus, status information is omitted from the examples.

The Virtual Machine Identifier Register (VMID) is also illustrated in the examples. Perhaps, more accurately it should be called a VCSID but the name VMID is kept for historical reasons. The value of the VMID identifies a particular VCSCB (or set of nested VCSCB's in the recursive case) and defines the map (or maps) that we have been calling f.

Example 1 - Type I VCS, Single Processor, R-B memory map

In this example, we illustrate some simple instruction executions on a virtual computer system implemented as a Type I VCS with single processor and relocation and bounds memory map. Figures 4-25 and 4-26 show part of the system base used by the VMM and part of the system base of VM1, the first level virtual machine (number 1). The dashed lines in Figure 4-25 indicate the segments of process P1 in the real machine, i.e. the VMM. The dashed lines in Figure 4-26 indicate the segments of process P1



(a) IC 1|200

(b) ϕ (1|200) = 1200

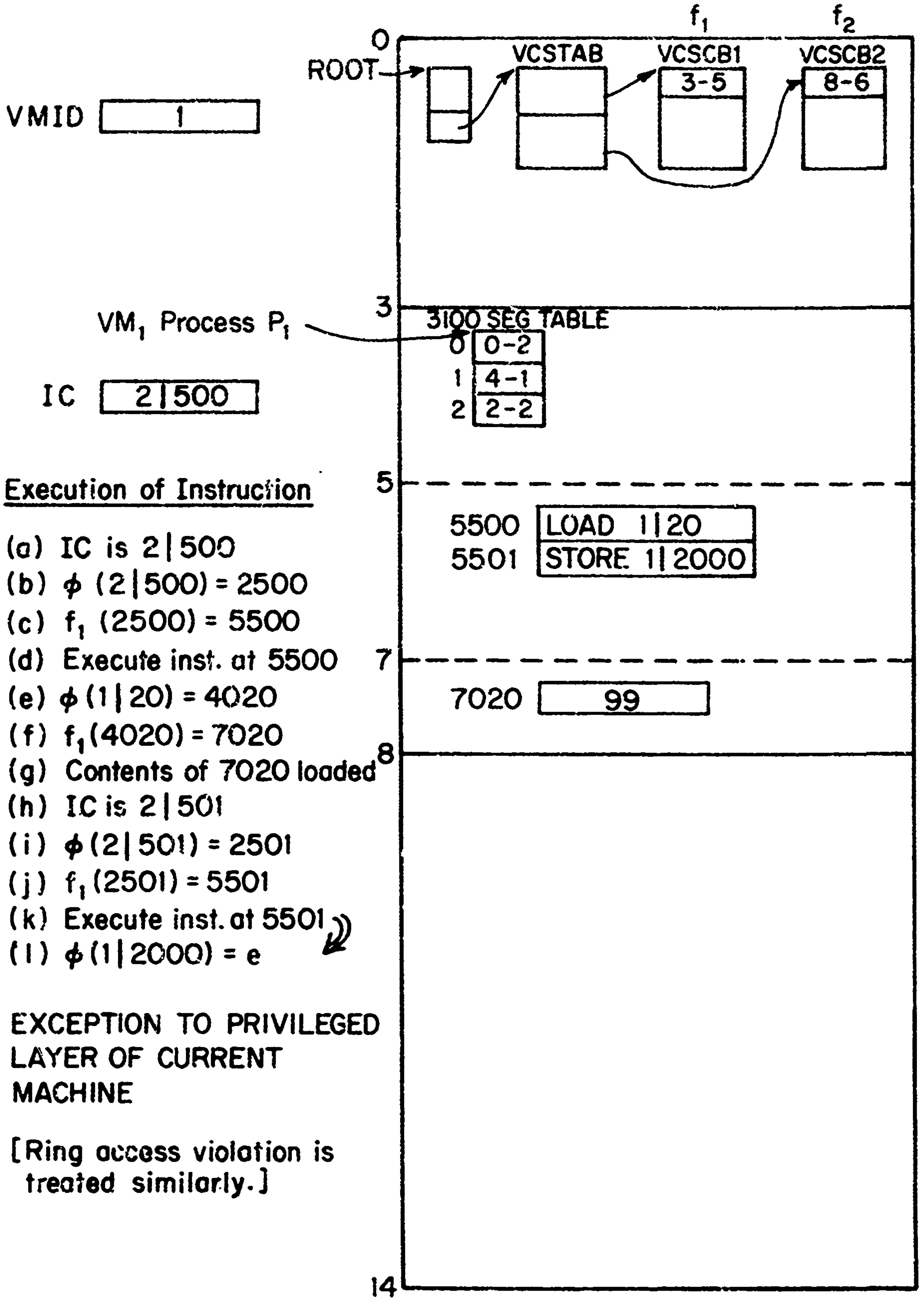
(c) Execute inst. at 1200

(d) ϕ (2|100) = 2100

(e) Loads 1 into the VMID register which was previously null. All CPU registers stored in VCSCB1 get loaded. VM1 is now active.

EXAMPLE 1 : THE LVMID INSTRUCTION

FIGURE 4-25



Execution of Instruction

(a) IC is 2|500
(b) ϕ (2|500) = 2500
(c) f_1 (2500) = 5500
(d) Execute inst. at 5500
(e) ϕ (1|20) = 4020
(f) f_1(4020) = 7020
(g) Contents of 7020 loaded
(h) IC is 2|501
(i) ϕ(2|501) = 2501
(j) f_1(2501) = 5501
(k) Execute inst. at 5501
(l) ϕ(1|2000) = e

EXCEPTION TO PRIVILEGED LAYER OF CURRENT MACHINE

[Ring access violation is treated similarly.]

EXAMPLE 1 : EXECUTION AND EXCEPTION

FIGURE 4-26

in the VM. The dark lines in Figure 4-26 indicate the VM's memory relocation and bounds.

Figure 4-25 illustrates the initiation of a VM by the VMM. Since process P1 of the real machine is running, the RPW in the system base is P1 [although this is not shown explicitly in the figure] and the processor's IC is 11200. This segmented address means segment 1, word 200.

The execution of instructions is traced in the commentary to the left of Figures 4-25 and 4-26. We will elaborate on the commentary for this example. The other examples can be followed without as much elaboration.

The instruction counter value must be mapped via the process map ϕ from a process name, 11200, into a resource name, 1200. This is done as a result of the segment table lookup. Since VMID=NULL, execution is on the real machine, and the resource name, 1200, is a real resource. Consequently, we must fetch and attempt to execute the instruction located at physical address 1200. This is the LVMID. A similar address calculation is carried out for the instruction's operand. The operand obtained is '1' and that value is loaded into the VMID register. Consequently, VM1 is activated and the information stored in the VCSCB, i.e. memory map, processor maps, etc., gets loaded.

Figure 4-26 picks up the action when the VMID is 1. Since process P1 of VM1 is now running, the RPW of the VM's system base is P1 [although this is not explicitly shown in the figure] and the processor's IC is 21500.

The instruction counter value must be mapped via the process map ϕ from a process name, 21500, into a resource name, 2500. This is done as a result of the segment table lookup. The resource name developed is a virtual name since the VMID=1. Therefore, it must be mapped via the VMmap f into its corresponding real equivalent. Since the VMmap's memory component is of form R-B, the real equivalent is obtained by checking the bounds and adding the relocation. Both are expressed in, say, 1000 word units. Thus, f1(2500)=5500. Consequently, we must fetch and attempt to execute the instruction located at physical address 5500. This is the LOAD instruction. A similar address calculation is carried out for the instruction's operand [see steps (e)-(g)] and the next instruction's address [steps (h)-(k)].

When we try to develop the address of the STORE instruction's operand, i.e. 1:2000, we cause an exception. On application of the process map ϕ, we discover that 2000 exceeds the bounds of segment 1 in process P1 (in VM1). Thus, an exception occurs to, say, a privileged layer of the current machine. This privileged procedure may be in process P1 or it might be in some other process that gets dispatched as a result of the exception. [See the semaphore discussion, below.] In any case, this event occurs without the knowledge or concern of the VMM. The VMID is still '1' as all VM local exceptions that occur are handled automatically.

In a typical IV generation system, similar exceptions and local exception handling might occur for access violations or dynamic segment linking. These are events that will likely occur with higher frequency than in earlier III generation systems and it is important to have them handled with the minimum of intervention.

Figure 4-27 is a continuation of Example 1. The intent of this extract is to illustrate possible referencing of the system base in the virtual machine. Since IV generation systems will be unlikely to have an absolute addressing mode, segmented addresses must be used to address the system base even though these entries may use absolute addresses as pointers. [We have illustrated the access to the system base using conventional LOAD and STORE instructions. This, of course, assumes that the layered ring mechanism is being used to control access to the system base. A special instruction might be used instead.]

The VMID contains '2' so VM2 is currently in execution. Assume that the active process, P1, is running in a privileged layer and thus may access the system base. Since P1's segment table lies in segment 0, the segment descriptor for segment 2 may be addressed as 0:102. Both steps (g) and (n) which LOAD and STORE a system base value, respectively, occur directly without interruption. This is because the VMmap f is distinct from the system base being modified. In a III generation software implemented VMmap f, both steps (g) and (n) would have to fault and be simulated. As we shall see, below, the VP [51] permits (g) to execute directly but faults on (n).

[Below, we will discuss potential problems arising from the

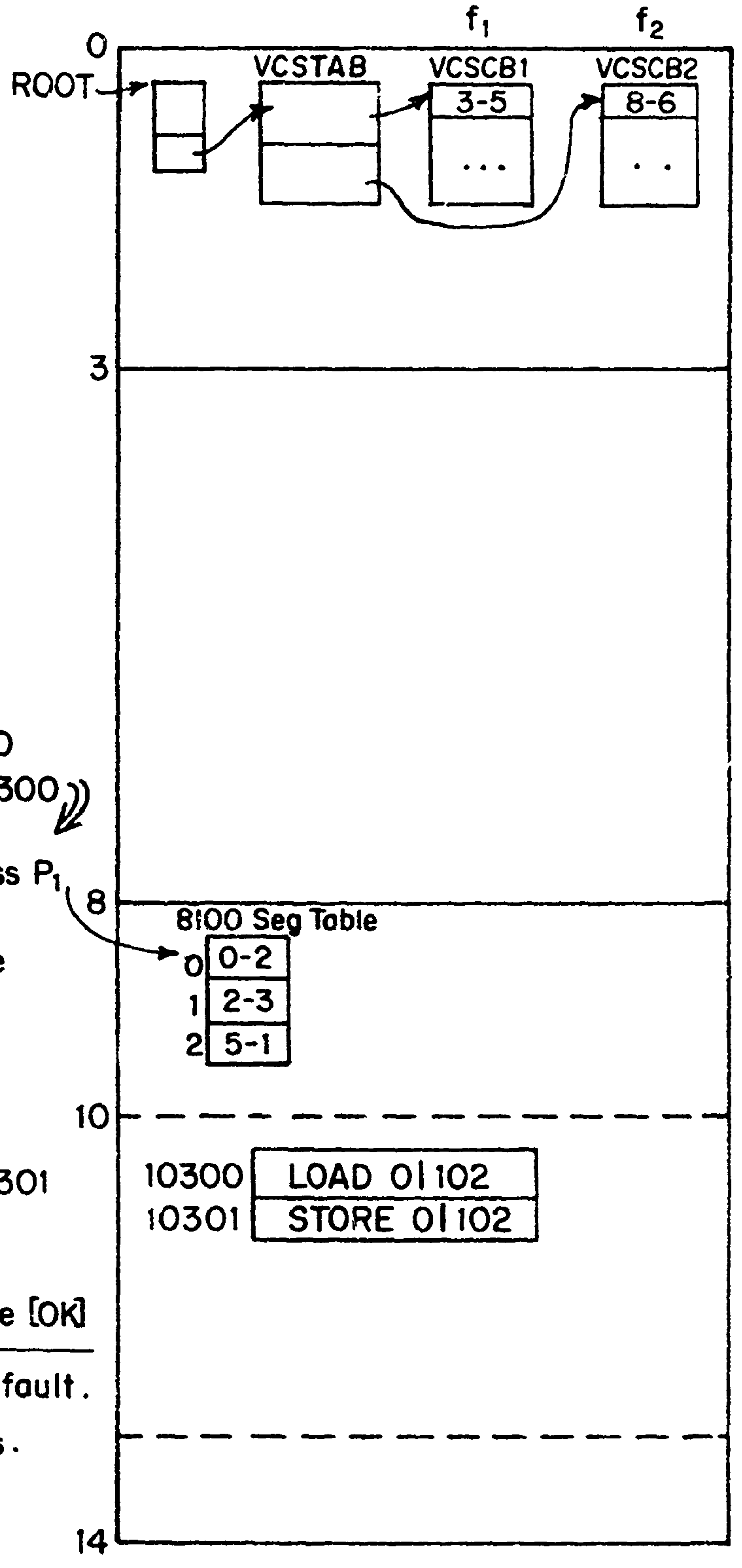
VMID $\boxed{2}$

IC $\boxed{1 | 300}$

(a) IC 1|300
(b) ϕ (1|300) = 2300
(c) f_2 (2300) = 10300
(d) Execute inst. at 10300
(e) ϕ(0|102)=0102
Process P_1
(f) f_2(0102)=8102
(g) Reads the seg. t[illegible]le
value $\boxed{5-1}$ [O[illegible]
(h) IC 1|301
(i) ϕ(1|301) = 2301
(j) f_2(2301) = 10301
(k) Execute inst. at 10301
(l) ϕ(0|102) = 0102
(m) f_2(0102)= 8102
(n) store into seg. table [OK]

In III gen., (g) + (n) fault.

In VP, only (n) faults.



EXAMPLE 1 : EXECUTION AND SYSTEM BASE UPDATE

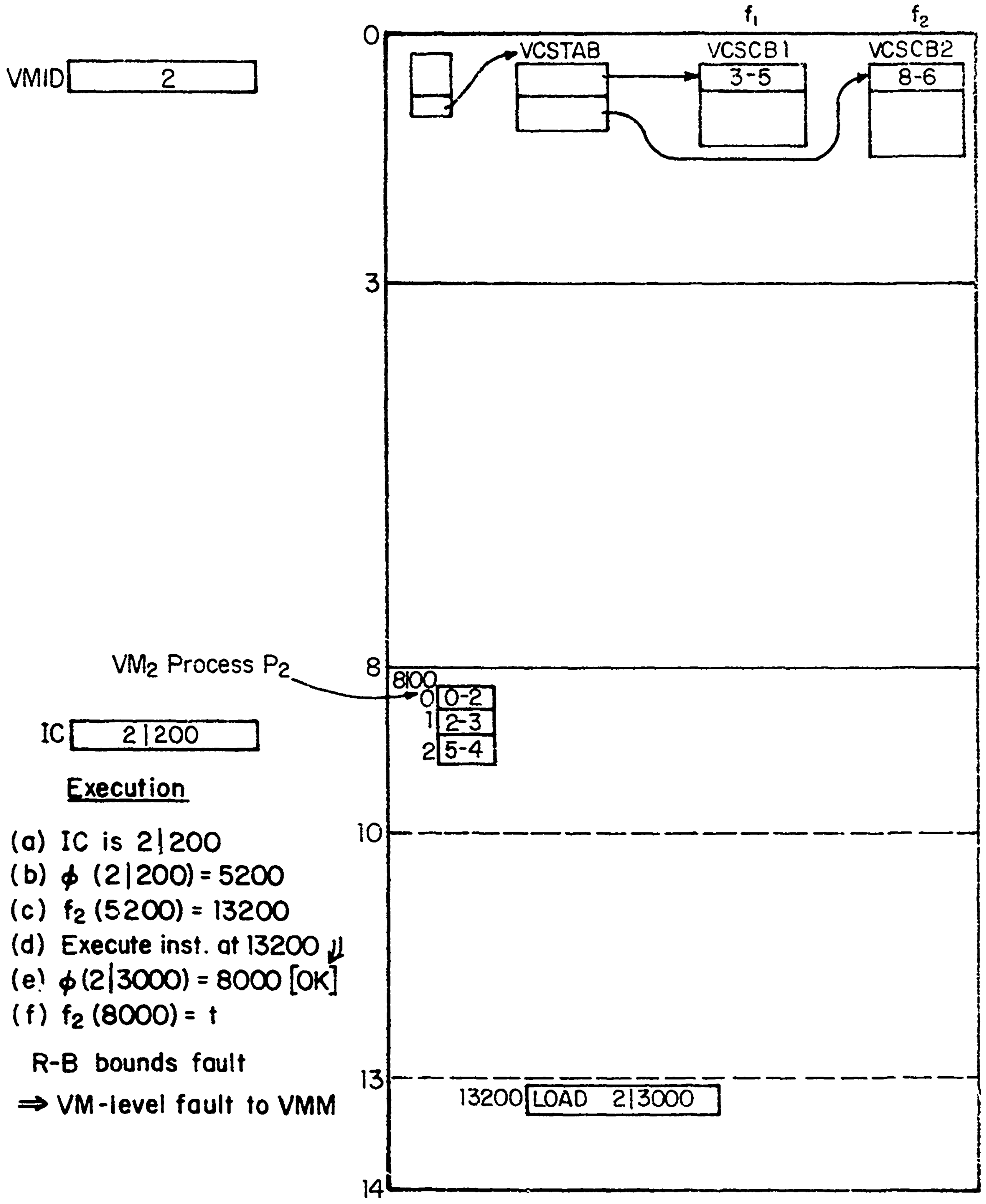
FIGURE 4-27

use of an associative memory which remembers the most recent map compositions. Under certain circumstances, unpredictable results may occur in both the real machine and in the virtual machine.]

Figure 4-28 is a further continuation of Example 1. VM2 is currently running and from f2, i.e., VMCB2, we see that only 6000 words of memory are allocated to it. The active process, P2, has a segment table which indicates that 9000 words of memory are allocated to the process. No validity check is performed when the system base entries are stored; they are verified only when used.

Thus, in the example, in step (e) application of the process map $\phi(213000)=8000$ does not cause a process exception. However, application of the VMmap, f2(8000) causes a VM-fault to the VMM. The VM-fault faults to the machine whose VMID is formed by dropping the low-order syllable from the current VMID. In this case, the fault is to the null identifier, or the real physical machine.

In order to speed up the operation of the process map ϕ, the real machine will likely use an associator. Similarly, operation of the virtual machine requires an associator to speed up the composed map $f \circ \phi$. [See Section 4.5.] Figure 4-29 sketches these corresponding associators and values that would be stored after execution of the instructions of Figure 4-26. [Running on



Execution

(a) IC is 2|200
(b) ϕ (2|200) = 5200
(c) f_2 (5200) = 13200
(d) Execute inst. at 13200 ⇓
(e) ϕ (2|3000) = 8000 [OK]
(f) f_2 (8000) = t

R-B bounds fault
⇒ VM-level fault to VMM

EXAMPLE 1: EXECUTION AND VM-FAULT
FIGURE 4-28

Associator — Real Machine — Process P_1

After Figure 4-26 Steps

	Seg #	Address	Validity, etc.
(b)	2	2000	
(e)	1	4000	

Associator — Virtual Machine 1 Process P_1

After Figure 4-26 Steps

	Seg #	Physical Address	Validity, etc
(c)	2	5000	
(f)	1	7000	

EXAMPLE 1: ASSOCIATOR
FIGURE 4-29

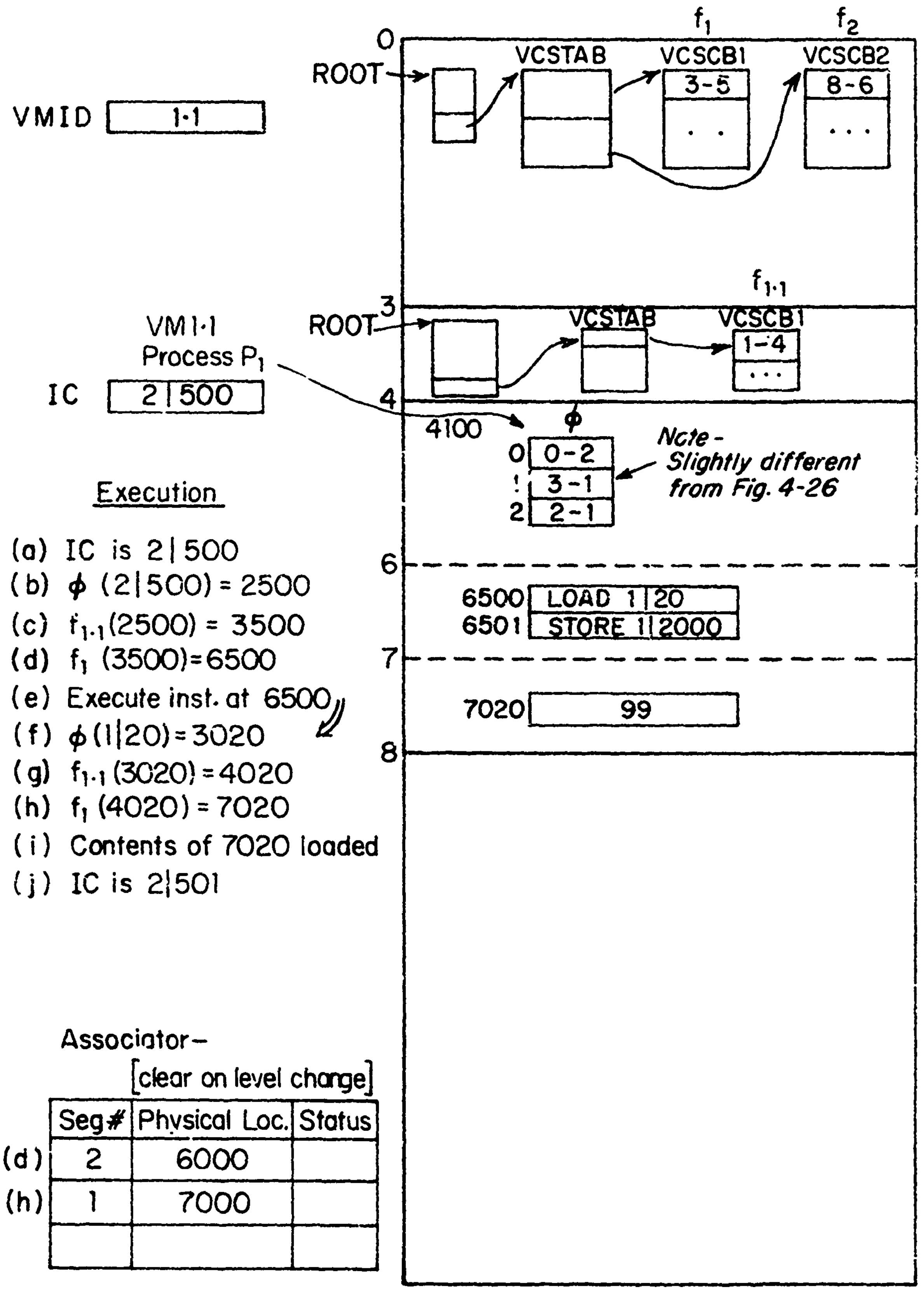
the real machine there would be no steps (c) and (f).]

In both cases we assume that a change in the Running Process Word, RPW, causes the associator to be cleared of entries. In the VM case, we further assume that a change in VMID does likewise. In any event, a change by an active process to its map ϕ or certain other parts of the system base, e.g., RPW, may cause unpredictable results (in the real or virtual system).

Example 2 - Recursion Added to Example 1

In Example 2, we illustrate the effect of recursion on our previous example. The two level-one virtual machines are the same as before. However, in this example, a VMM runs on VM1. Figure 4-3. illustrates the VMmap fragment in the system base of VM1. After VM1 executes the 'LVMID 1' instruction, the VMID row contains '1.1'. The brackets on the right side of the figure indicate the amount of memory resource owned by the real machine, VM1, and VM1.1.

When we pick up the action, process P1 of VM1.1 is executing a sequence of instructions similar to that of Example 1. [Note that P1's segment table entries are slightly different in this example.] After applying the process map ϕ(2153]) we must apply f1.1 and then f1 until we finally map into physical resource names. The corresponding associator is also illustrated in Figure 4-30. Once again, segment numbers are mapped into physical locations. This time, the entries represent f1 o f1.1 o



Execution

(a) IC is 2|500
(b) ϕ (2|500) = 2500
(c) $f_{1\cdot1}$(2500) = 3500
(d) f_1 (3500)=6500
(e) Execute inst. at 6500
(f) ϕ(1|20)=3020
(g) $f_{1\cdot1}$(3020)=4020
(h) f_1 (4020) = 7020
(i) Contents of 7020 loaded
(j) IC is 2|501

Associator–
[clear on level change]

	Seg#	Physical Loc.	Status
(d)	2	6000	
(h)	1	7000	

EXAMPLE 2: EXECUTION AND ASSOCIATOR

FIGURE 4-30

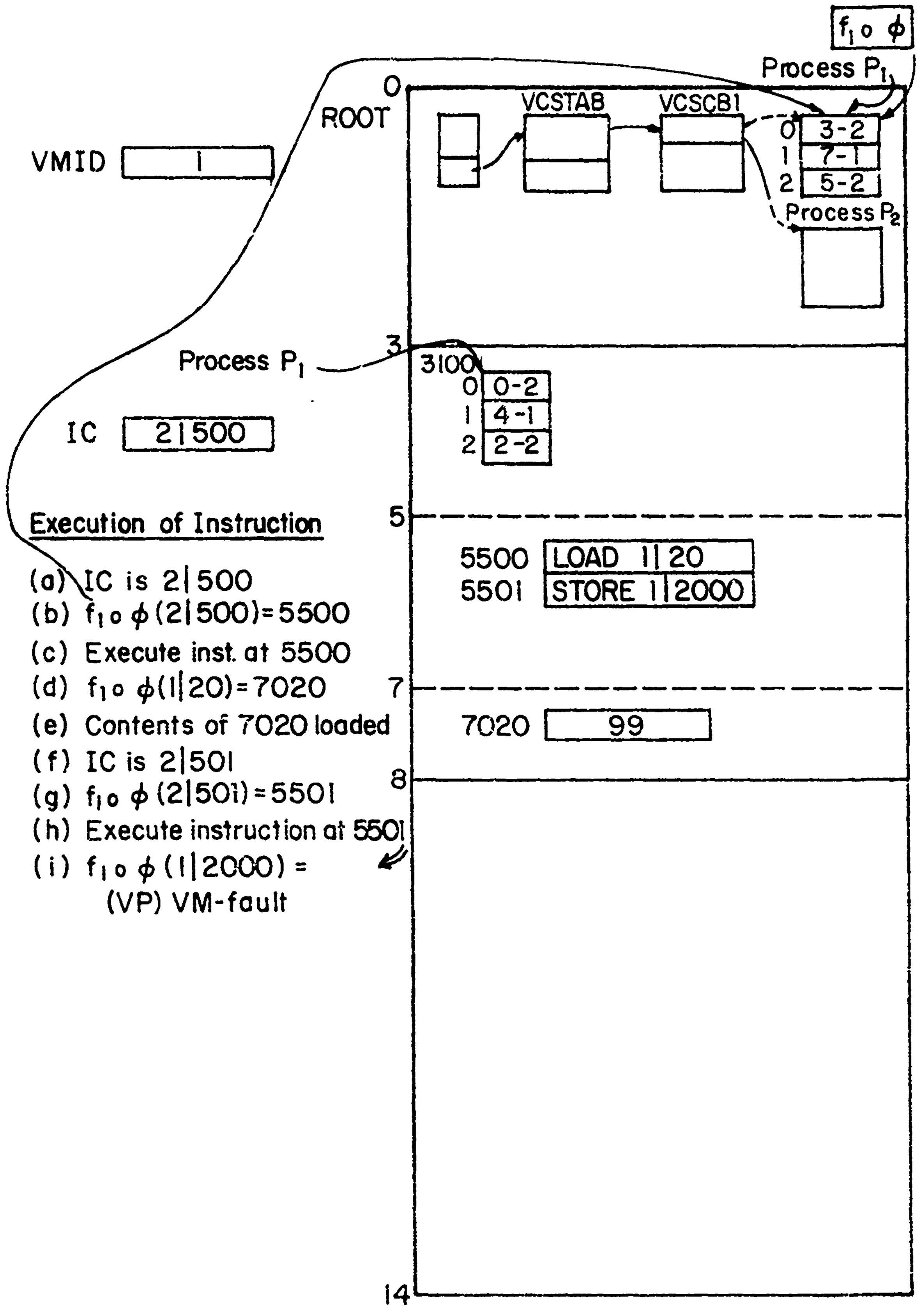
ϕ. Note that the associator need not be larger in the R-B recursive case than in the real machine or in the single level R-B case.

While we do not illustrate the other fragments from Example 1, they behave similarly. The process exception has a local effect and does not alert the VMM. The system base may be referenced in LOAD or STORE instructions without causing a VM-fault. Finally, a VM-fault by VM1.1 is directed to VM1.

Example 3 - Similar to Example 1 illustrating VP

In the next example, we contrast how the Venice Proposal [single level - no recursion] might work for the instruction sequence fragment of Example 1. Again, we assume two fixed partitions of the main memory for the virtual machines. Since the VP does not explicitly provide the map f [as the HV does] f must in effect be provided in software. What the VP does provide, however, is the composed map $\phi t = f \circ \phi$ for all processes of all virtual machines. Therefore, the VCSCB's of the VP do not provide the memory, processor maps etc. Instead each VCSCB points off to the ROOT of a complete copy of the translated system base. Thus, in particular, there exists a path from a VCSCB to its processes' PCB's and corresponding segment tables.

In Figure 4-31, we abbreviate this structure by showing a dotted line pointer from the VCSCB to segment table. The contents of the segment table for process P1 of virtual machine 1



Execution of Instruction

(a) IC is 2|500
(b) $f_1 \circ \phi(2|500)=5500$
(c) Execute inst. at 5500
(d) $f_1 \circ \phi(1|20)=7020$
(e) Contents of 7020 loaded
(f) IC is 2|501
(g) $f_1 \circ \phi(2|501)=5501$
(h) Execute instruction at 5501
(i) $f_1 \circ \phi(1|2000) =$
(VP) VM-fault

EXAMPLE 3: VP EXECUTION

FIGURE 4-31

gives the absolute physical location of the segments, e.g. the map $\Phi t = f \circ \Phi$ [for level 1 VM].

Thus, in step (b) of the instruction execution, $f1 \circ \Phi$ is applied to 21500 yielding the physical location 5500 directly.

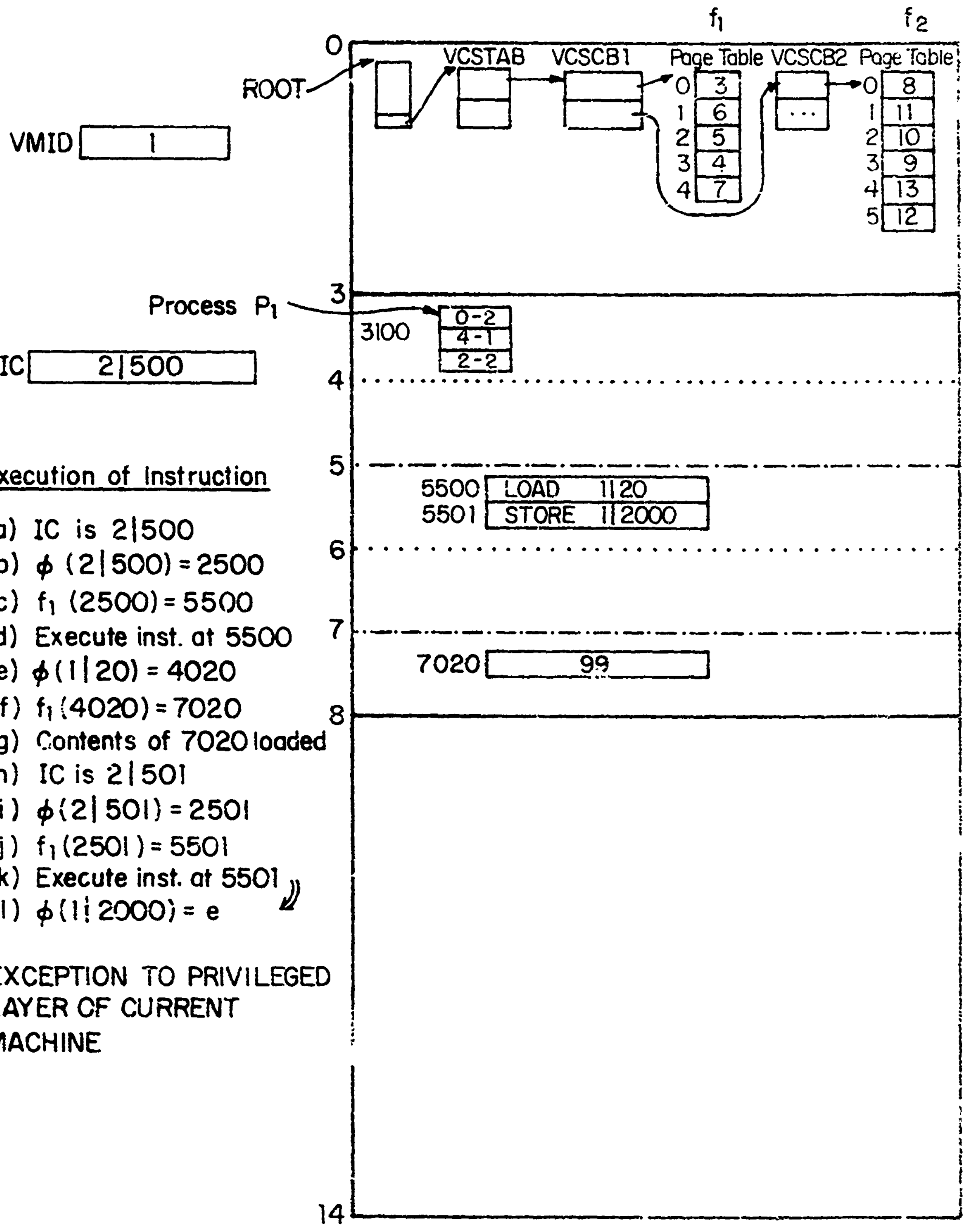
This ex: ole does not indicate reading or writing system base instructions, exceptions or faults since we have already discussed these cases in the earlier text.

Example 4 - Type I VCS, Paged Memory Map

The next example, Figure 4-32, illustrates how a paged memory map might be added to the VCSCP. We assume that the page block size is 1000 words [which is also the minimum segment unit]. We do not illustrate fictitious resources or page faults. Although the map f1 is somewhat different from Example 1, its application produces identical results. See steps (a)-(l) for confirmation.

In the paged memory map, the associator plays a significant performance role. Figure 4-33 illustrates the possible form of this associator. Whereas the R-B associator did not differ structurally from the 'real' associator [although the addresses stored were different], the paged associator has somewhat different structure and will likely be larger than the others [for performance considerations].

Note, once again, that this a sociator is different from that used in the IBM 360/67 [68] or the Honeywell 645 [34,86]



Execution of Instruction

(a) IC is 2|500
(b) ϕ (2|500) = 2500
(c) f_1 (2500) = 5500
(d) Execute inst. at 5500
(e) ϕ(1|20) = 4020
(f) f_1(4020) = 7020
(g) Contents of 7020 loaded
(h) IC is 2|501
(i) ϕ(2|501) = 2501
(j) f_1(2501) = 5501
(k) Execute inst. at 5501
(l) ϕ(1|2000) = e

EXCEPTION TO PRIVILEGED LAYER OF CURRENT MACHINE

EXAMPLE 4: EXECUTION OF PAGED VCS

FIGURE 4-32

Associator

After Figure 4-32 Steps

	Seg # - Page #	Physical Location	Status
(c)	2 -- 0	5000	
(f)	1 -- 0	7000	

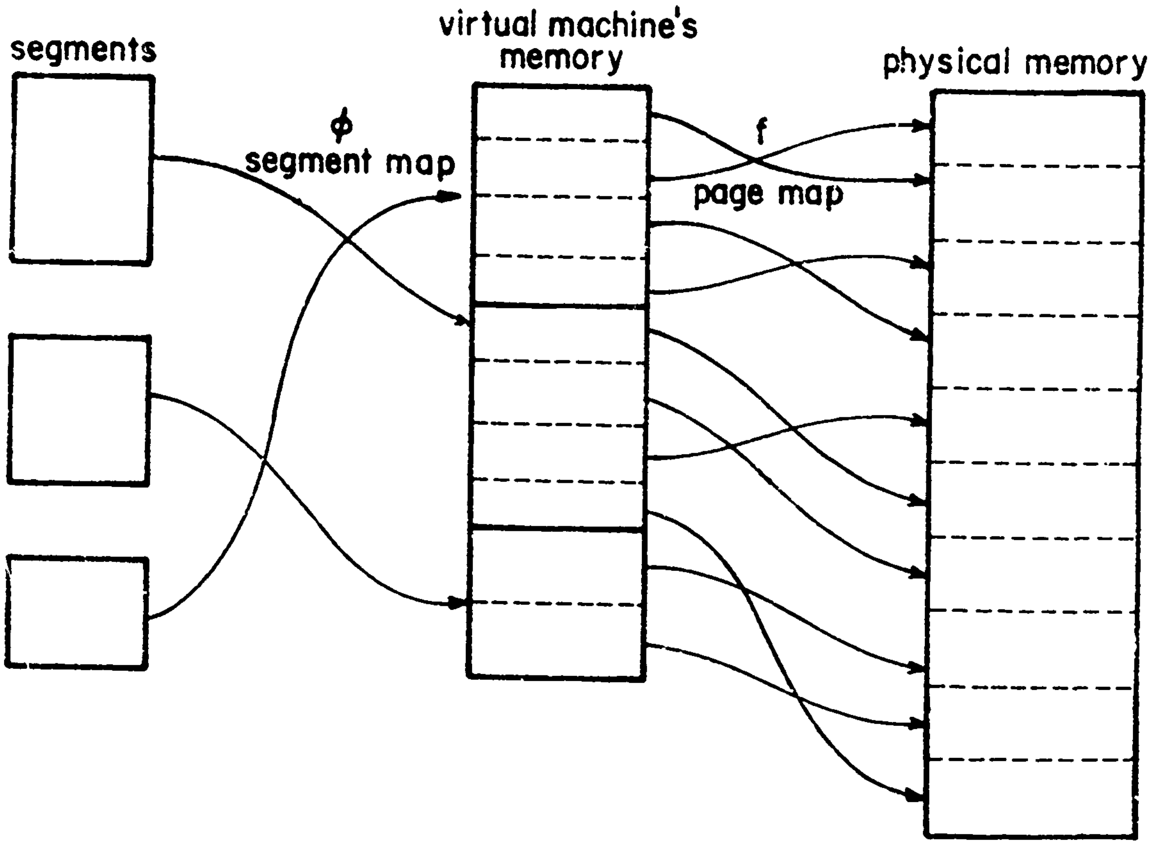
EXAMPLE 4: PAGED ASSOCIATOR
FIGURE 4-33

etc. since the f and ϕ map structure is different. The use of paging in Example 4 is for memory resource management only and does not contain any of the process structure normally found in paging maps. In particular, s[illegible] Figure 4-34 for an illustration of the difference. Thus, while [illegible]ntation, a software visible construct, is found in the ϕ map, paging, a resource allocation mechanism, is best added to a system as part of an f-map. The use of this approach implies that complete operating systems will be runnable without modification in a paged environment. This result should be contrasted with OS/360 and TSS/360 [10]. See Section 4.8, below.

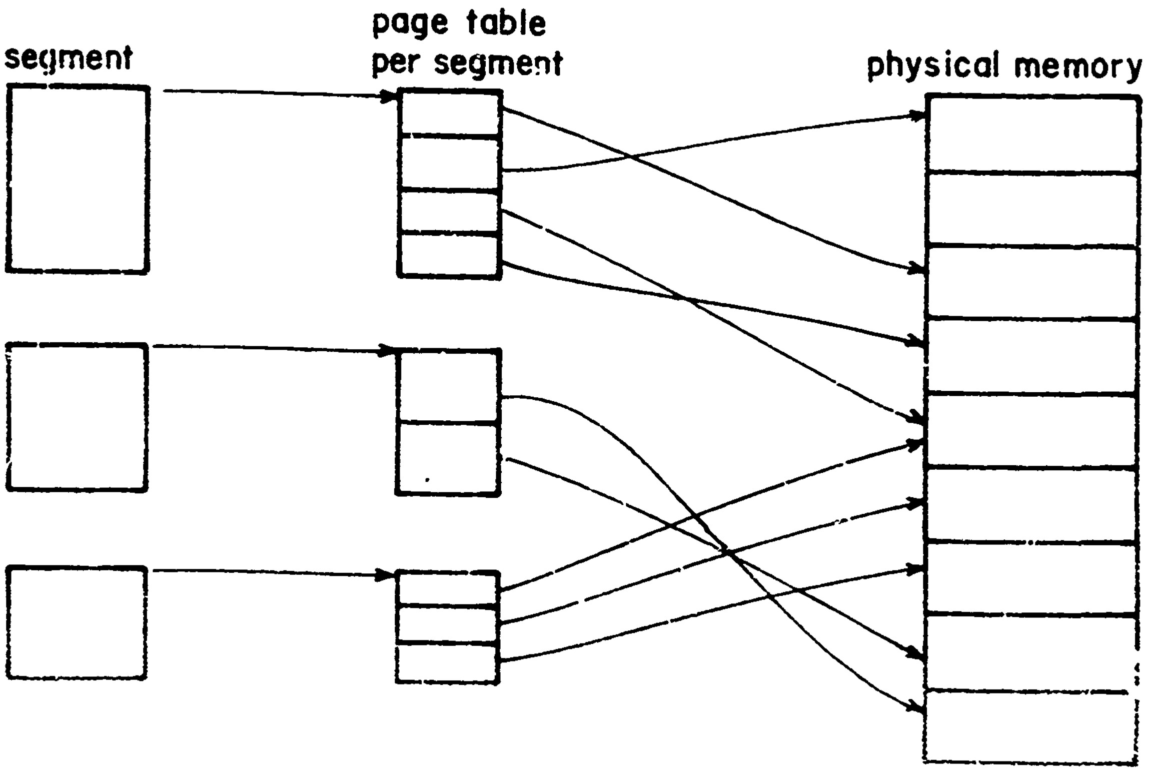
Example 5 - Example 4 with Recursion

In Example 5, Figure 4-35, we add recursion to the paged memory map of Example IV. As can be seen, this example is very similar to the recursive R-B example of Figure 4-33. The principal apparent difference is the form of the associator. However, because of locality considerations, as discussed above [in Section 4.6] acceptable performance should result. Thus, we have shown that [unlike VP] recursion does not introduce additional complexity, even in the case of a complicated memory map such as paging.

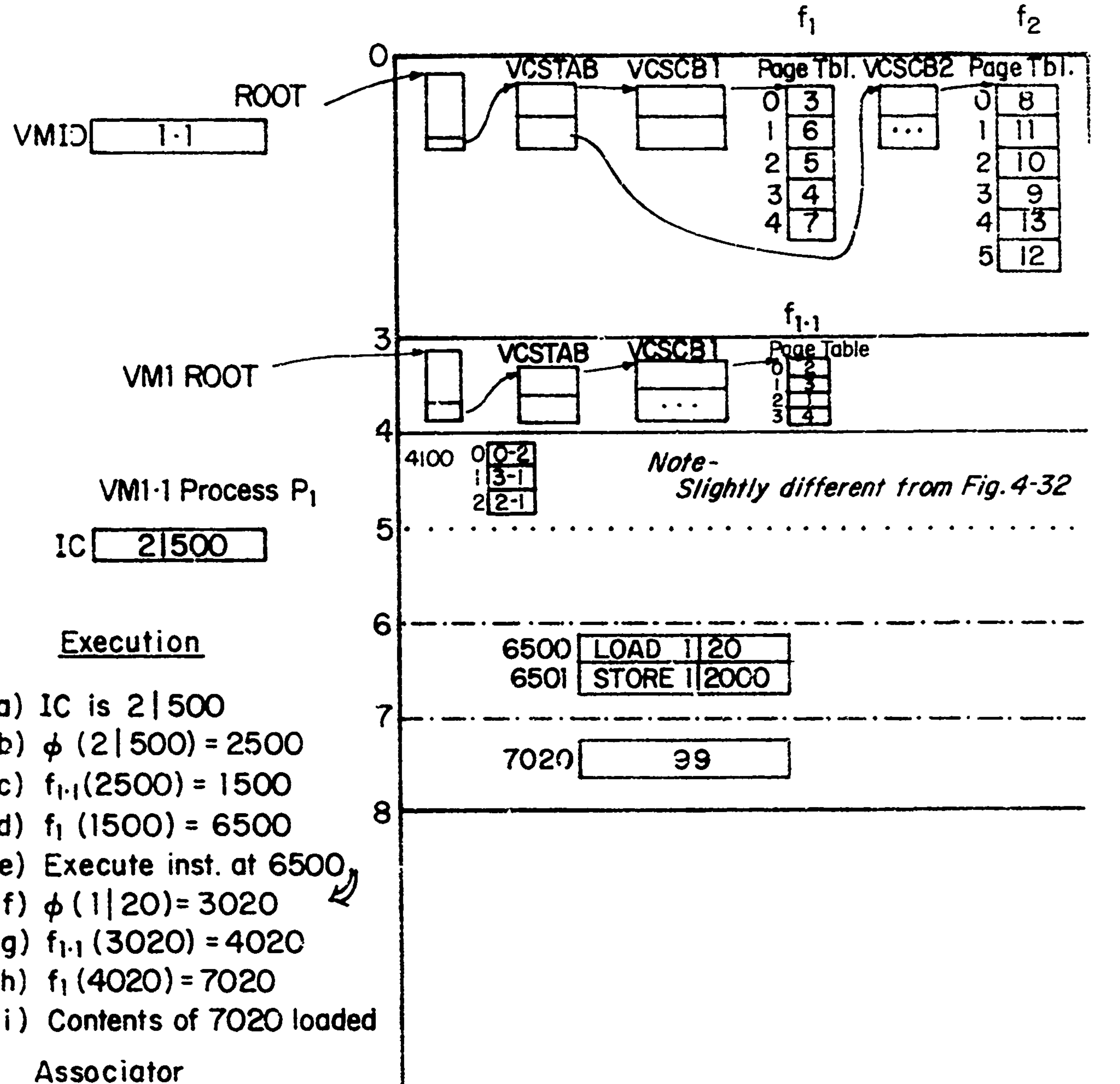
HV with paged VM map: One page map (f) for all segments.



645, 360/67: One page map per segment



HV vs. 645, 360/67



Execution

(a) IC is 2|500
(b) ϕ (2|500) = 2500
(c) $f_{1\cdot 1}$(2500) = 1500
(d) f_1 (1500) = 6500
(e) Execute inst. at 6500
(f) ϕ (1|20) = 3020
(g) $f_{1\cdot 1}$ (3020) = 4020
(h) f_1 (4020) = 7020
(i) Contents of 7020 loaded

Associator

Step	Seg# and Page #	Phys. loc.	Status
(d)	2-0	6000	
(h)	1-0	7000	

EXAMPLE 5: EXECUTION AND ASSOCIATOR

FIGURE 4-35

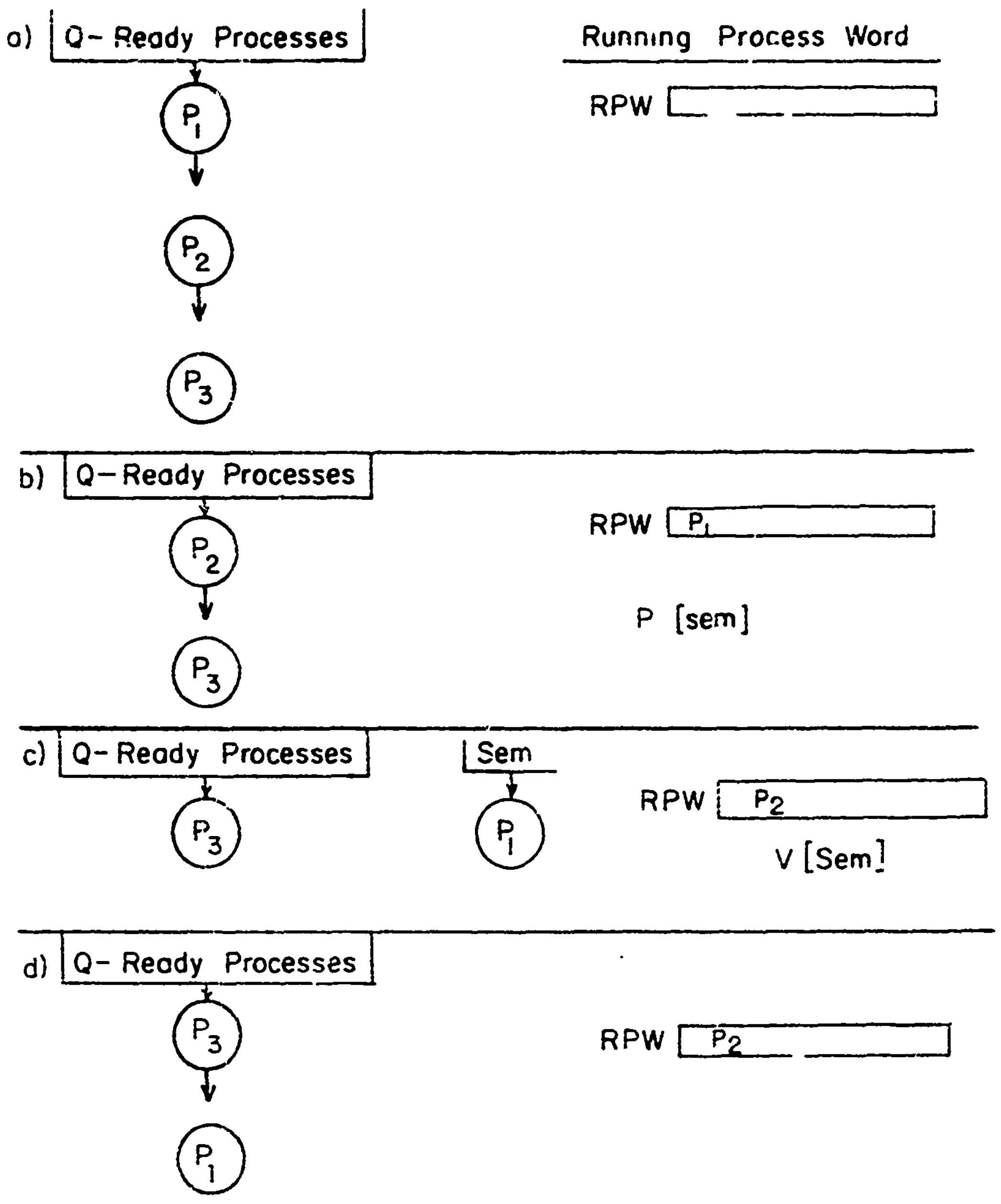
Example 6 - Semaphores in a Virtual Machine

The intent of this example is to further demonstrate that the virtual system base may be used directly in most instruction executions without the intervention of the VMM. We will illustrate how firmware dispatching and operations on semaphores might work in a virtual machine. [We will assume the reader is familiar with semaphores and Dijkstra' P and V operations [41]. This is crucial to the operation of a IV generation VM since IV generation 'Jobs' can be assumed to be made up of a number of cooperating sequential processes [87].

The previous examples of this section have illustrated that paged or other memory maps in the VMCB introduce no additional complication that cannot be solved as in the R-B case. Thus, we will illustrate semaphores with an R-B map.

First, we illustrate semaphores and process dispatching on the real machine. In Figure 4-36, frame (a) shows the queue of ready processes pointed to by the system base. Each (circle) entry in the queue represents a process link for some active process in the system. These links contain the process name and, so, effectively a pointer to the corresponding PCB. The queue is organized, presumably, in priority order, although we do not illustrate or concern ourselves with the priority field.

In frame (b), the first process is removed from the Q-Ready processes by the firmware dispatcher. Its (process) identifier is set in the Running Process Word, RPW. Let us assume that process P1 executes a P(sem) and the semaphore count of sem is 0. Then P1 is blocked and put on the queue of semaphore sem. The



SFMAPHORES AND PROCESS QUEUES

FIGURE 4-36

next process, P2 in the figure, is made active [Frame (c)]. Process P2 executes a V[sem] which removes P1 from the semaphore queue and returns it to the Q-Ready processes [Frame (d)].

All of these manipulations of the system base are normally carried on entirely by the firmware [79]. We will show that this can be the case in the virtual system as well.

Figure 4-37 provides a more detailed picture of the system base illustrating the semaphore tables and process links. Each PCB points to a semaphore table which defines the semaphores for that process. If the semaphore exists, then the semaphore table entry is a pointer to the semaphore header for that semaphore. In Figure 4-37, the semaphore 'sem' is called semaphore number 3 by process P1 and semaphore number 4 by process P2.

The semaphore header contains the count and, if negative, a pointer to the top of the semaphore queue. The pointer is given as a relative displacement to the origin of the Active Process Link Table. Subsequent queue entries are linke[illegible] together similarly. [Note that this is one possible very simpl[illegible] for semaphores that may be used in this illustrat[illegible] machines. We do not claim it to be the mos[illegible] organization.]

Figure 4-37 represents the system base bef[illegible] executes the P[sem] instruction. The executi[illegible] of P[sem[illegible] the firmware to apply the Φ a[illegible] f maps as indicate in th[illegible] commentary. Finally, Figure 4-38 shows the system [illegible] after the P[sem] instruction. Execution of the subsequent V[sem] instruction by process P2 may be traced in a similar manner.

VMID

1

RPW

1

IC

2|504

Execution of Instruction

(a) IC is 2|504

(b) ϕ (2|504) = 2504

(c) f (2504) = 5504

(d) Fetch Inst. at 5504

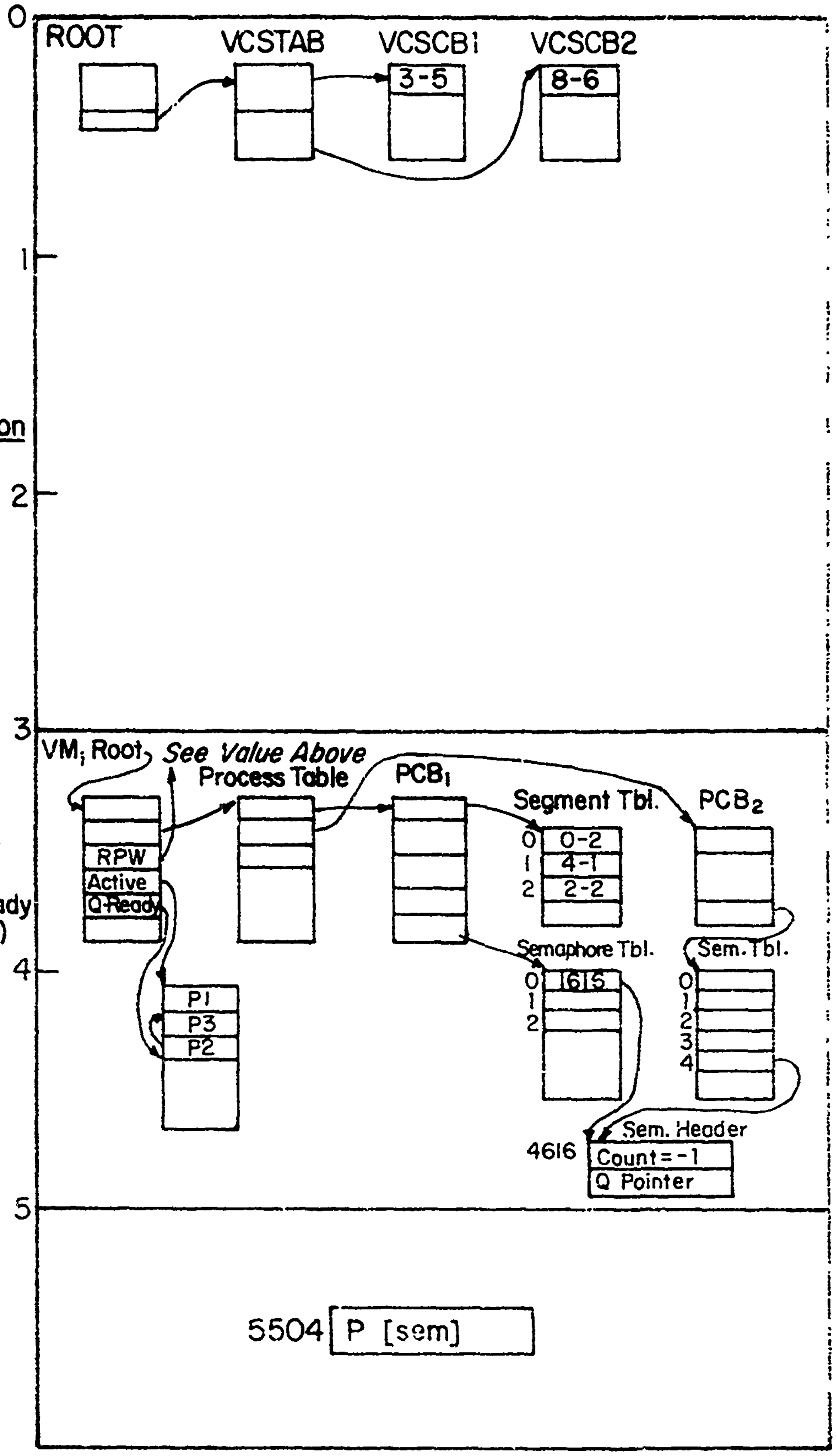
(e) ϕ (sem.0) = 1616

(f) f (1616) = 4616

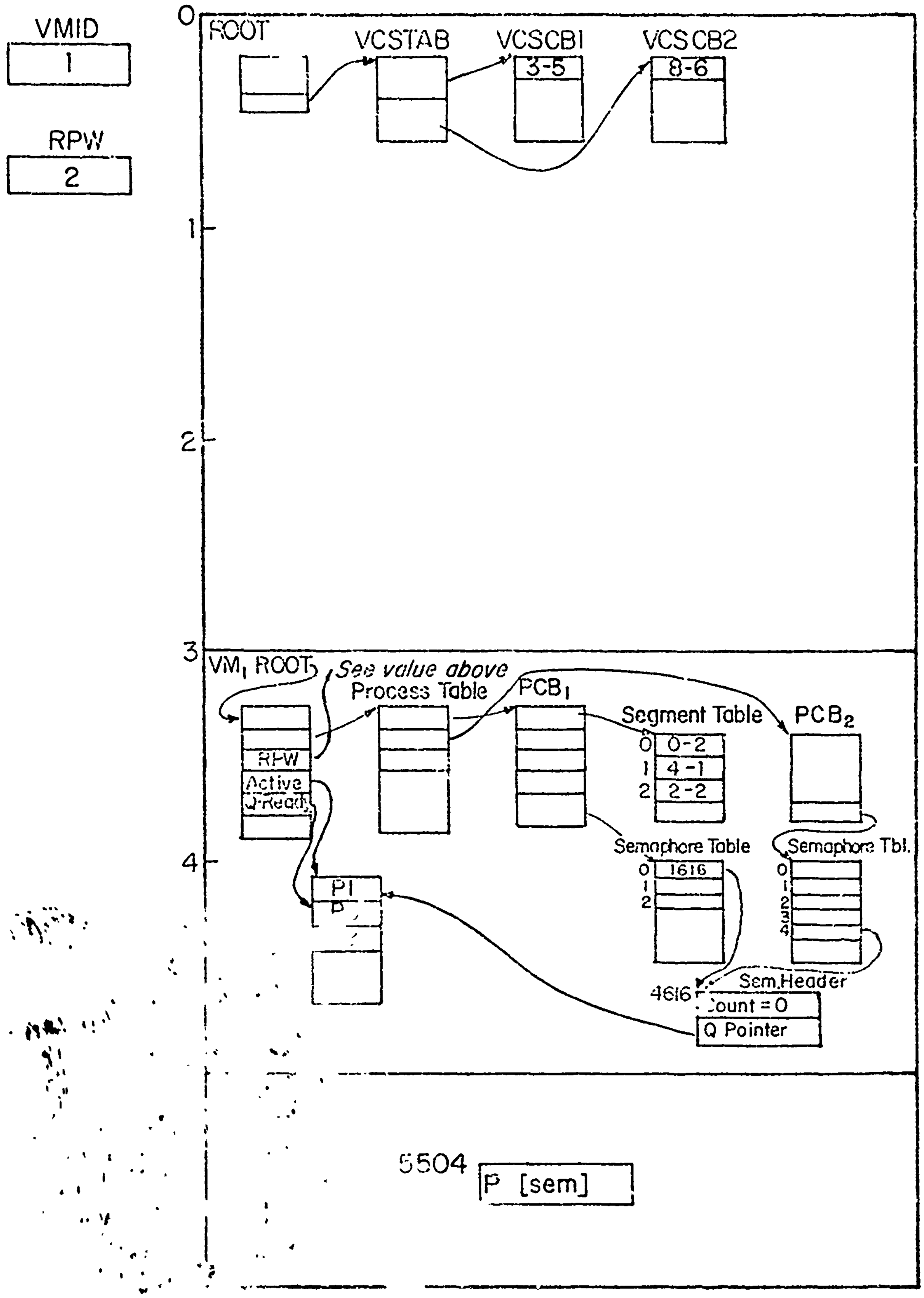
(g) Decrement contents of 4616 by 1

(h) Since contents of 4616 < 0 place process 1 in semaphore sem queue

(i) Dispatch next Q-ready process (process 2) by placing 2 in the RPW



EXAMPLE 6: SYSTEM BASE DETAIL FOR SEMAPHORES
FIGURE 4-37



SYSTEM BASE DETAIL AFTER P [sem]
FIGURE 4-38

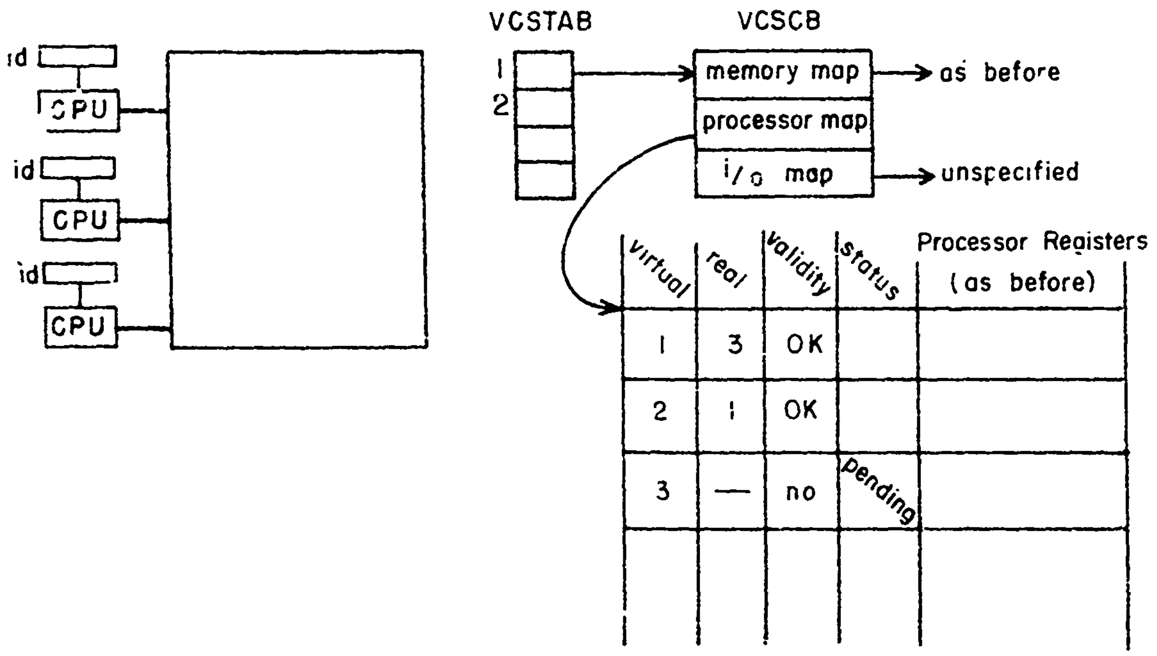
Example 7 - Type I VCS, Multiple Processors

This example illustrates the extension introduced by multiple real and virtual processors in a computer system. For the purposes of the example we assume that all processors are identical CPU's. [Clearly this work can be extended directly to non-timing-dependent I/O processors as well [15,17,96].]

Figure 4-39 shows the modification to the VCSCB. The processor map now points off to a virtual processor table, each entry of which gives the (instantaneous) map between virtual and real processors. Also the LVMID instruction must be expanded to include a second operand providing the processor number as well as the VCS number.

Thus, when a physical processor, say, processor 3 executes 'LVMID 1,1', the VMID for the processor is set and real processor 3 is set in the processor map as corresponding to virtual processor 1. Later, when virtual processor 1 executes 'SIGNAL 2' the instruction executes directly by looking up the corresponding real processor value. When 'SIGNAL 3' is attempted, since the map value is invalid, a pending bit is set. [A VM-fault may also occur to allow the VMM to schedule virtual processor 3 ...]

Note that this example illustrates just one way that the f map may be introduced for multiple processors. The virtual-real processor index is a kind of page map and the dispatching of the VCS without all processors assigned is an example of a fictitious



EXAMPLE 7: MULTIPLE PROCESSOR'S PROCESSOR MAP

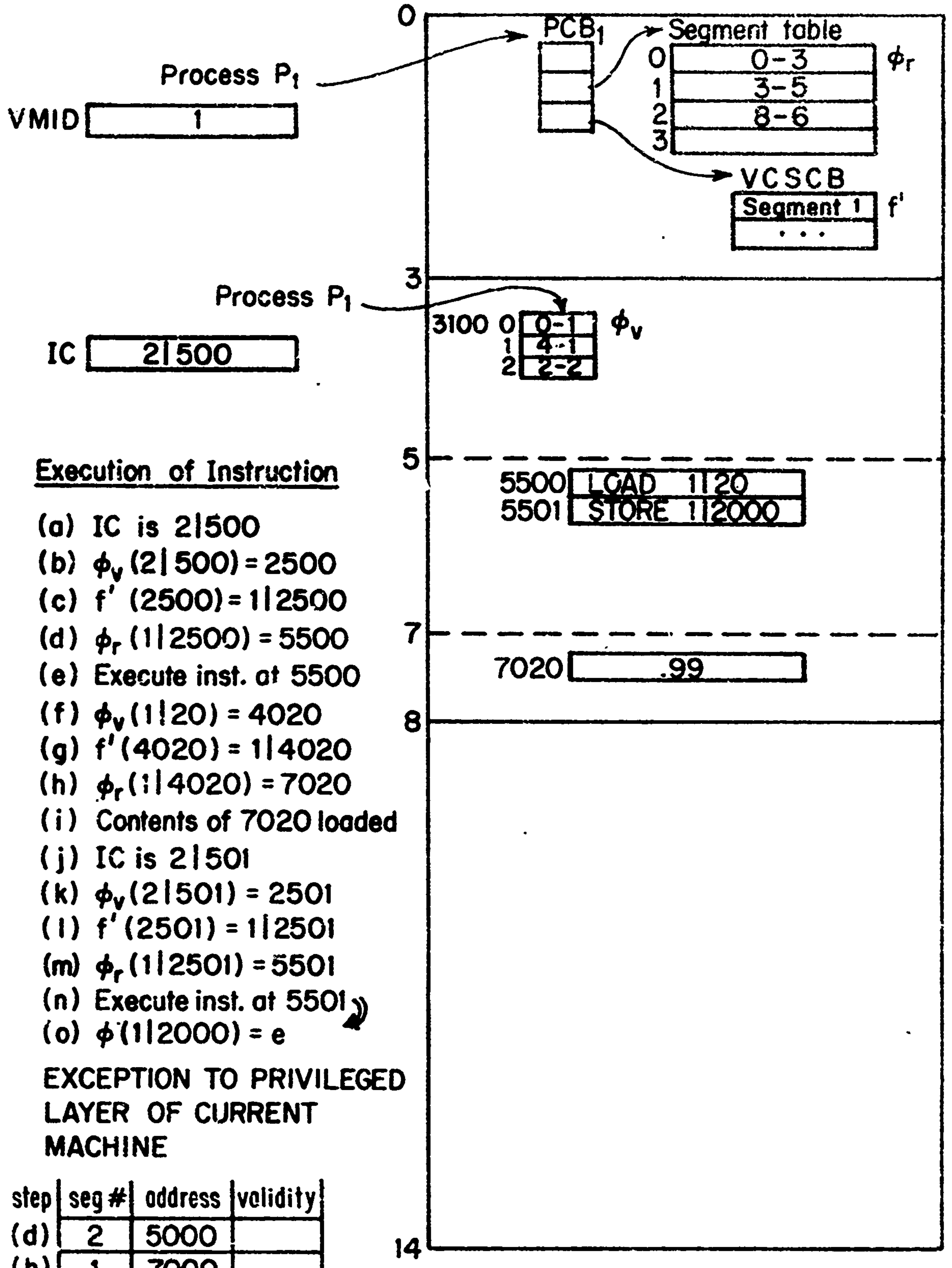
FIGURE 4-39

resource. We can imagine other kinds of processor f maps. For example, with a relocation-bounds processor VMmap, an induced real processor number is obtained by adding the relocation to the virtual processor number after checking that the bound is not exceeded. One can imagine a system organized around this map requiring all virtual processors to be assigned by start-up, i.e. no fictitious (dynamic) resources.

Example 8 - Type II VCS, Single Processor

In this example, we give one possible interpretation of a IV generation Type II VCS. To show the operation, we return to the instruction sequence from Example 1. Our system base now has the VCSCB hanging off the PCB (of some particular privileged process). We have the same memory, processor, etc. components of the map. However, now virtual resource names are mapped into process names. Thus, the memory map illustrated takes virtual memory addresses into segments. In our example, we map addresses into displacements within segment 1 of process #1. Steps (b)-(d) successively apply the process map of [illegible] machine, ϕv, the virtual machine map, f°, and [illegible] process map of the real machine, ϕr. See Figure 4-49. The a [illegible] be of the same form as that used in Example 1.

As seen before, in Chapter [illegible] a number of advantages to running a Type II VMM. [illegible] ration system, a Type II VMM permits the predominant [illegible] ting system to run



Execution of Instruction

(a) IC is 2|500
(b) ϕ_v(2|500) = 2500
(c) f' (2500) = 1|2500
(d) ϕ_r(1|2500) = 5500
(e) Execute inst. at 5500
(f) ϕ_v(1|20) = 4020
(g) f'(4020) = 1|4020
(h) ϕ_r(1|4020) = 7020
(i) Contents of 7020 loaded
(j) IC is 2|501
(k) ϕ_v(2|501) = 2501
(l) f'(2501) = 1|2501
(m) ϕ_r(1|2501) = 5501
(n) Execute inst. at 5501
(o) ϕ(1|2000) = e

EXCEPTION TO PRIVILEGED LAYER OF CURRENT MACHINE

step	seg #	address	validity
(d)	2	5000	
(h)	1	7000	

EXAMPLE 8: TYPE II VCS

FIGURE 4-40

without degradation due to the VMmap. Furthermore, it simplifies the construction of the VMM since the host operating system takes over the allocation of real resources. In this example, virtual memory is being mapped into segments of a process running on the real machine. Therefore, the allocation of memory for these segments and the response to segment-absent exceptions may be handled by the standard operating system memory allocation procedure.

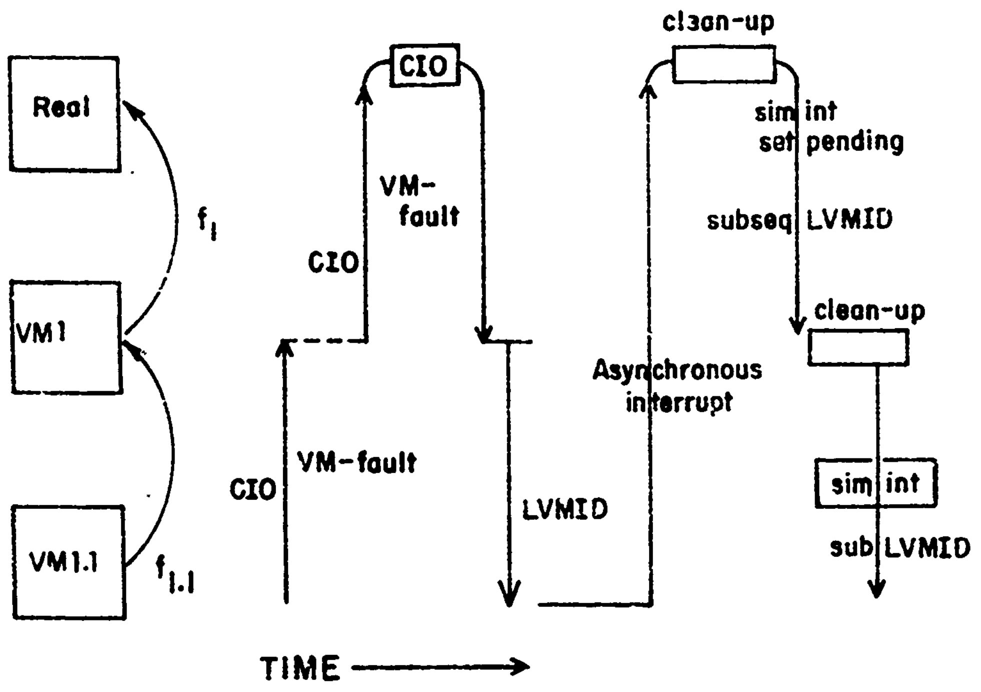
4.7 SOME PRACTICAL PROBLEMS OF IV GENERATION HV

There are a number of practical problems that prevent the hardware virtualizer, as described, from providing the complete implementation of a IV Generation VCS. Some of these difficulties, e.g., time, are intrinsic to virtual computer systems. Other difficulties, such as the ad hoc unformalized time-dependent I/O process, are technological and may pass as we gain greater understanding of computer systems architecture. Until such time as these problems are overcome, we can still rely on III generation, software VMmap techniques. We will illustrate how an interim solution can be developed for I/O.

The difficulty with I/O lies chiefly with its control, not with memory or device mapping which may be accomplished via the VCSCB. The interim solution we opt for directs I/O completion interrupts to the real machine. We make the Connect (Start) I/O instruction (and all other I/O instructions) absolutely privileged such that it may only be issued by the physical machine and attempted execution by a VM causes a VM-fault to the VMM. As with III generation systems, the VMM translates the request and issues the I/O request itself. On the I/O termination, an interrupt occurs to the VMM running on the absolute machine. Regardless of the VM being run at the time of the interrupt, its status may be saved, and control is returned to the virtual machine (by the physical machine) after interrupt processing. The VCSCB is expanded to include an I/O interrupt pending bit which is set by the VMM. Execution of the LVMID instruction to start up a VM while the pending bit is set for that VM, causes an I/O interrupt to occur to that VM.

While the asynchronous I/O interrupt goes to the absolute physical machine, the connect instruction (CIO) VM-fault is still a conventional HV fault up one level to the VMM. This is an improvement over the III generation handling of I/O in which the connect (start) instruction (via the software f) traps to the physical machine. The real trap handler must reflect the trap back down to the appropriate level VM, running the VMM. Thus, even for software I/O map, in the IV generation HV, recursion is much easier and more natural than in III generation systems. See Figure 4-41.

Another suggestion for I/O control can be derived from Lauer and Snow [76]. Basically, they suggest allowing the virtual machine to issue I/O instructions subject to the device map in the VCSCB. I/O completion interrupts then come in to the real machine which passes the information down the VMM chain via so-called "spurious interrupts."



INTERIM I/O PROPOSAL

FIGURE 4-41

4.8 II AND III GENERATIONS REVISITED

The work of Chapter 4 has been directed toward introducing the HV into IV generation systems. We have assumed that IV generation systems define an architecture with a process map ϕ implemented in firmware. We have then expanded the design to form a new system which incorporates a level invisible firmware VMmap, as well.

We have assumed that ϕ is a rather complex map. However, the principles we have developed are completely independent of the complexity of ϕ. Thus, it becomes possible to examine other computer architectures in which the maps f or ϕ may be of different form from those considered. The important properties which must be preserved, however, are that ϕ corresponds to the software visible structure <u>within</u> a level of a virtual machine whereas f corresponds to the map <u>between</u> two levels of virtual machines. We will consider two representative examples.

4.8.1 f=R-B,ϕ=identity

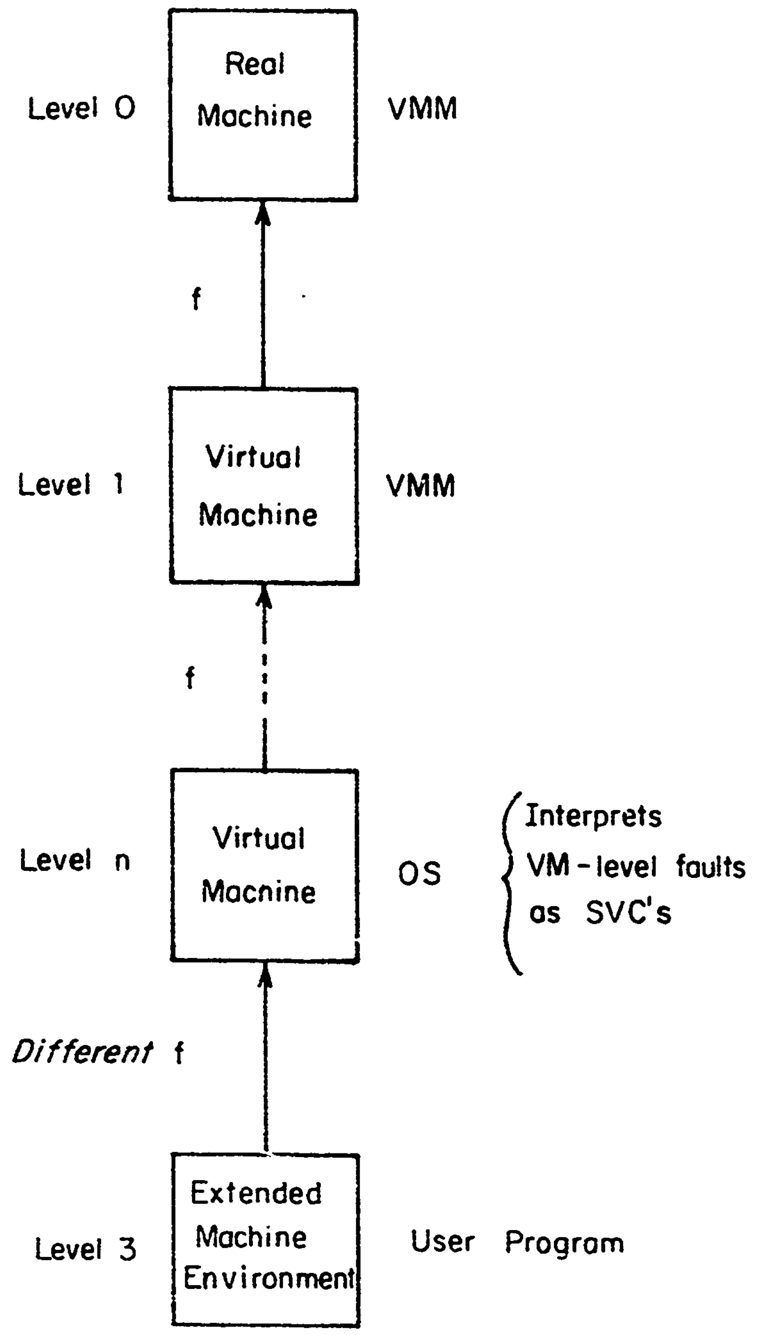
If we let ϕ=identity map, then we have a computer design without intra-level structure. There is only one mode, i.e. "Supervisor state is not necessary" [76]. There is no memory mapping within a level. The level structure looks very much like a first or second generation design of the sort found in a DEC PDP-8 class minicomputer.

The introduction of an R-B f-map allows us to develop virtual machines for this system. [As in Section 4.5.] Because of the lack of structure in each machine level, it may be desirable to allow programs to communicate between levels by causing a VM-level fault with a conventional bounds violation code. This now defines a programming environment in which the VM-level fault is used to call on supervisory services, much as the "diagnose" instruction in CP-67 [2,23,65]. In Figure 4-42, the VMM running at level n-1 produces a VM at level n. At level n, we run a "two-level operating system" OS [in the figure], which produces this enhanced environment at level n+1. The system will work as long as level 0 to level n-1 each run the VMM. This is true because of VM-recursion and level invisibility of the f-maps.

The "f=R-B,ϕ=identity" machine is essentially that proposed by Lauer and Snow [76]. They observe:

> It is more general than most existing hardware in the sense that it naturally encourages a hierarchical structure without imposing artificial boundaries or an omnipotent supervisor. But it lacks some of the important features of modern systems, particularly segmented virtual memories and a suitable parameter passing mechanism ... [76]

The introduction of intralevel structure, e.g. process model and segmentation, can best be accomplished using the techniques developed earlier in Chapter 4. However, the "ϕ=identity" machine may still be a useful system to construct, particularly in conjunction with a line of minicomputers. In this context, both f=R-B and f=paging could lead to interesting results.



"f = R-B, ϕ = IDENTITY" MACHINE

FIGURE 4-42

4.8.2 f=paging,ϕ=supervisor/problem-state [no memory mapping]

The work of Chapter 4 indicates how paging could have been introduced into an IBM System/360 Virtual Computer System. Instead of paging being a part of the visible intra-level structure, it should have been introduced as part of a formal f-map. Under these circumstances, there would be no need to rely on the techniques discussed in Chapter 3, i.e., a "software VMmap". Privileged instructions would be able to execute directly [without the trap simulate dispatch sequence], intra-level exceptions would be directed to the privileged software at the current level, recursion would be vastly simplified, and the VMM, i.e. CP-67, would be greatly streamlined. Once again, a key point is that paging, a mechanism for resource allocation, should properly be part of the f-map, not the ϕ-map.

CHAPTER 5.

CONCLUSION

5.0 SUMMARY

The thesis has developed appropriate terminology, an empirical basis, and a model which represents the execution of processes on a VCS. The model features two maps: (1) a process map called ϕ which maps process names, e.g. segments, semaphores, process-id's, into resource names, e.g. memory locations, processor numbers, and (2) a virtual machine map (VMmap) f which maps virtual resource names into real resource names. The process map is strictly an intra-level map expressing a relationship within a virtual machine; the VMmap is an inter-level map expressing a relationship between (the resources of) two adjacent levels of (virtual) machines. Thus, the action of running a process on a VCS consists of running it under the composed map $f \circ \phi$. Application of the model has led to a clear interpretation of virtual machine recursion, Type II virtual machines, and other important properties of virtual machines.

The model also allows us to understand the implementation of III generation VCS's as the software construction of the VMmap f, utilizing parts of the hardware process map ϕ. However, the most important result of studying the model is that it leads directly to the design of a hardware virtualizer for proposed implementations of IV generation VCS's. The design has the characteristic that all process exceptions are handled directly within the executing VCS without software intervention. All resource faults (VM-faults) by a VCS are directed to its VMM on the real machine without knowledge of processes on the VCS. Fault handling, invocation and execution of VCS's works directly regardless of recursion. Furthermore, preliminary performance studies indicate that this design will provide performance of the virtual machine comparable to the real machine for the likely choices of the maps f and ϕ.

5.1 SUGGESTIONS FOR FURTHER RESEARCH

The development of the principles reported in this thesis has also pointed out a number of areas for future research. The model and suggested implementations of Chapter 4 have been directed largely toward virtualization of a single processor (without I/O) computer system. Example 7 of Section 4.6 sketches an approach to multiple processor configurations and Section 4.7 suggests an interim solution to I/O. However, there is still much work to be done in incorporating either of these features into the model and the hardware virtualizer. As Section 4.7 points out, progress in the treatment of virtual I/O is somewhat tied to progress in the treatment of real I/O. As long as timing dependencies and asynchronous interrupts exist in I/O systems, it will be difficult to treat the virtualization of I/O in a homogeneous manner with the rest of the system.

Another area for further development of architectural principles concerns machines with reloadable control store [97,98]. The work of the thesis has been directed to virtualization of interior decor, and, as part of the defintion of a VCS, we have assumed the existence of some "native mode." However, the whole concept of interior decor in future systems is likely to be extended by reloadable control store. The incorporation of such machines into the theory will have an important impact upon preparation and extension of code for the microprocessor.

The study of VCS's seems to provide an excellent meeting ground for a number of traditionally separate disciplines of computer science. One area, in particular, concerns the development of models which provide an interior decor level description of the execution of processes on a computer system. The empirical work of Chapter 3 has provided an essential foundation upon which to create a useful abstraction of program execution in a III generation VCS.

The thesis has been concerned with mechanisms for implementing VCS's, rather than specific policies for applying to them. Thus, we have not discussed issues of resource allocation, performance, or throughput as applied to the real or virtual systems. This includes the introduction of fictitious resources, i.e. demand paging, or particular multiplexing algorithms to be used in a VMTSS. It is quite important to develop models that will make possible reasonable resource allocation decisions. One aspect worthy of study concerns treatment of the process interval timers and the firmware process dispatcher. Should the virtual machine dispatcher be implemented as a firmware adjunct or will software be sufficient? How does this choice depend on the intended use of the system, i.e. many or few virtual machines. Another question concerns possible conflicts and counterproductive algorithms between different levels of virtualization. For example, if the ϕ-map is segmentation and the f-map is paging, then memory compaction by the virtual machine may be a sub-optimal policy. Another example might involve "double spooling"-- once by the virtual machine and once by the VMM. Performance models and studies will be a necessary part of future VCS theory and implementation.

APPENDIX A

TUTORIAL ON CP-67

A.0 Introduction

In this appendix, we shall illustrate the typical software implementation of virtual machines on third generation systems by reviewing the mechanisms used by CP-67 [9,32,54,85].

In order to faithfully produce a virtual 360, CP-67 must provide a virtual processor, virtual memory, and virtual I/O channels and I/O devices. The basic approach that CP-67 adopts is to handle high frequency events in an automatic manner with little, or no, overhead, while handling infrequent events in a less efficient manner. This notion leads CP-67 to a rather efficient handling of virtual processor and virtual memory. Virtual I/O events which occur at a much lower frequency may be processed with greater overhead.

CP-67 is able to simulate a 360/67 in both of its modes of operation. In one mode, the 360/67 behaves as a 360/65, a standard 360 without memory mapping. In the other mode, called by IBM "extended mode", the 360/67 uses a number of additional registers and instructions. In extended mode, the 360/67 has the ability to enter relocate mode. The 360/67's operation in extended mode, but not relocate mode, is very similar to the operation of a 360/65.

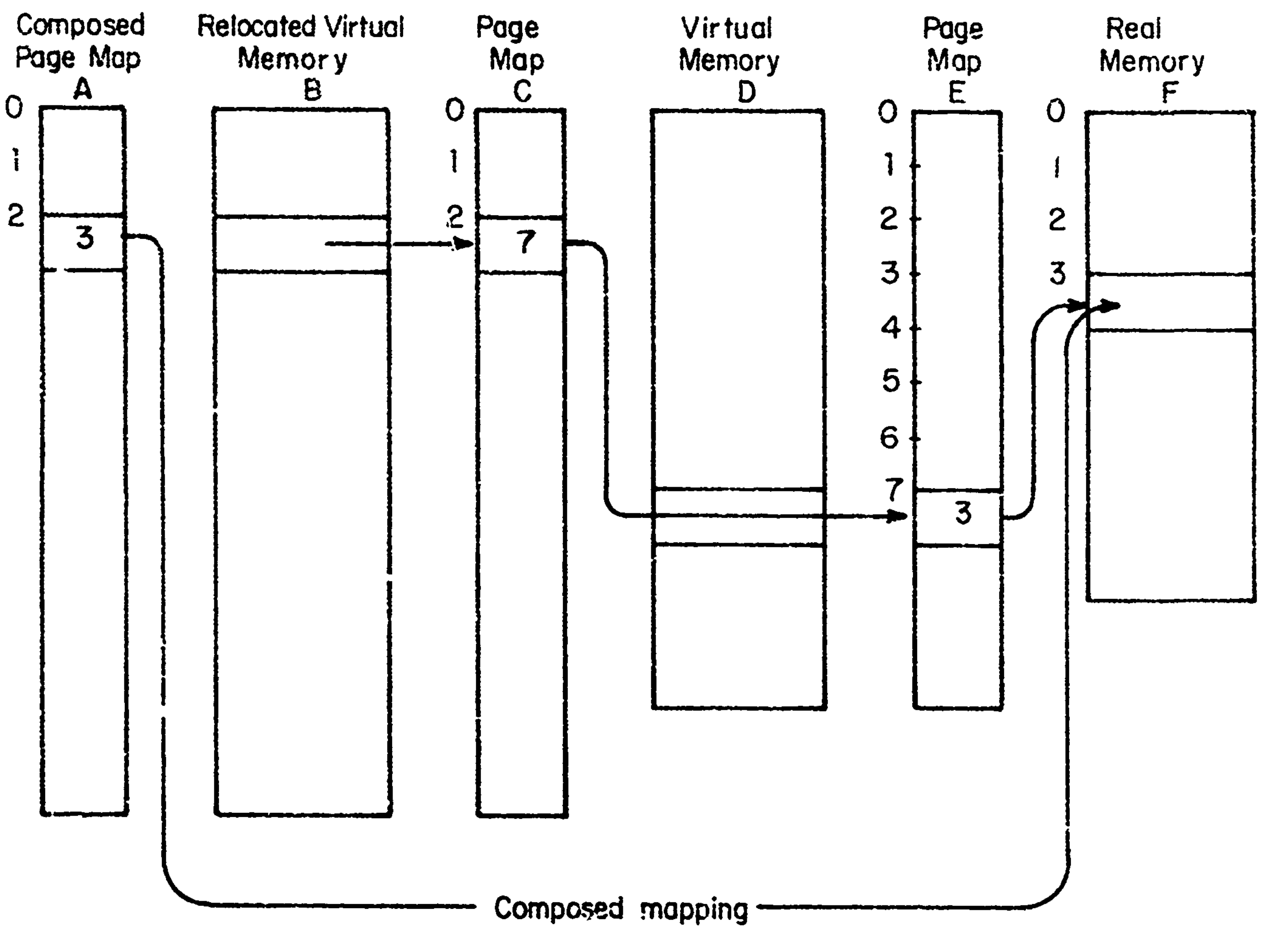
A.1 CP-67's Virtual Memory

The simulation of virtual memory for virtual machines is perhaps the easiest of CP-67's chores. The 360/67 has a memory relocation mechanism, called DAT (Dynamic Address Translation), which features a segmented-paged address space [112]. The 24 bit virtual address produced by the 360/67 in relocate mode is interpreted as a four bit segment number, an eight bit page number, and a twelve bit displacement within the page. Segment and page tables are located in core.

CP-67 always runs virtual machines in relocate mode. Thus, the paging mechanism theoretically permits virtual machines to address a memory of up to 2**24 or 16 million bytes. This virtual memory may far exceed the amount of physical memory actually available. In practice, however, for reasons of efficiency virtual computer systems are normally configured to have a significantly smaller virtual memory, usually less than 1 million bytes.

The paging mechanism may be used directly as long as the virtual machine, itself, does not attempt to execute in (virtual) relocate mode. If the virtual machine, e.g. 360/67, goes into relocate mode, it is necessary to map the relocated addresses of the virtual machine into their corresponding real addresses. This facility can be provided with reasonable efficiency via a software algorithm as described in Goldberg [54] and illustrated in Figure A-1.

In Figure A-1, the page map (C) provides the correspondence between addresses in the relocated virtual memory (1) and the



MAPPING RELOCATED VIRTUAL MEMORY TO REAL MEMORY
FIGURE A-1

(non-relocated) virtual memory (C). Thus, relocated virtual memory page 2 is mapped into virtual memory page 7. Similarly, the page map (E) provides the map from virtual memory (D) to real memory (F). Thus, virtual page 7 is mapped to real page 3. Since all addresses must ultimately be mapped into real addresses and the 360/67 hardware provides only a single level of page mapping [this example ignores the "segmentation" of the 360/67], it is necessary to use software to derive the composed page map (A) which maps directly from (B) to (F). Therefore, when the virtual machine that is illustrated executes in (virtual) relocate mode, relocated virtual page 2 is mapped into real page 3.

A.2 CP-67's Virtual I/O

The simulation of virtual I/O channels and devices is performed in a number of different ways by CP-67. Virtual machines may be configured with I/O devices that are realized as dedicated, spooled, or shared devices. A dedicated device is a device such as a tape drive or an entire disk pack which may be used by only one virtual machine at a time. Consequently, tape drives are attached to some virtual machine for a given duration. The virtual device address need not be the same as its physical address. A correspondence between the two is established at the time that the device is attached.

A shared I/O device is a physical device which may be simultaneously part of the configuration of a number of different virtual machines. A shared I/O device is divided among each of the different owners in such a way that each virtual machine believes that it alone owns the device, but that the virtual device is smaller than the actual physical device. An example of a shared device is a disk pack that has been divided among several virtual machines. These so-called mini-disks behave exactly like the ordinary IBM 2311 disks except that they have fewer cylinders. Another example of a shared I/O device is a transmission control unit, such as the IBM 2702, in which each virtual machine may have only a small number of the transmission lines into the device. Such sharing artifacts, like the mini-disk, are introduced for cost or "efficiency" considerations. They improve the sharing of large resources that might otherwise be ineffectively utilized.

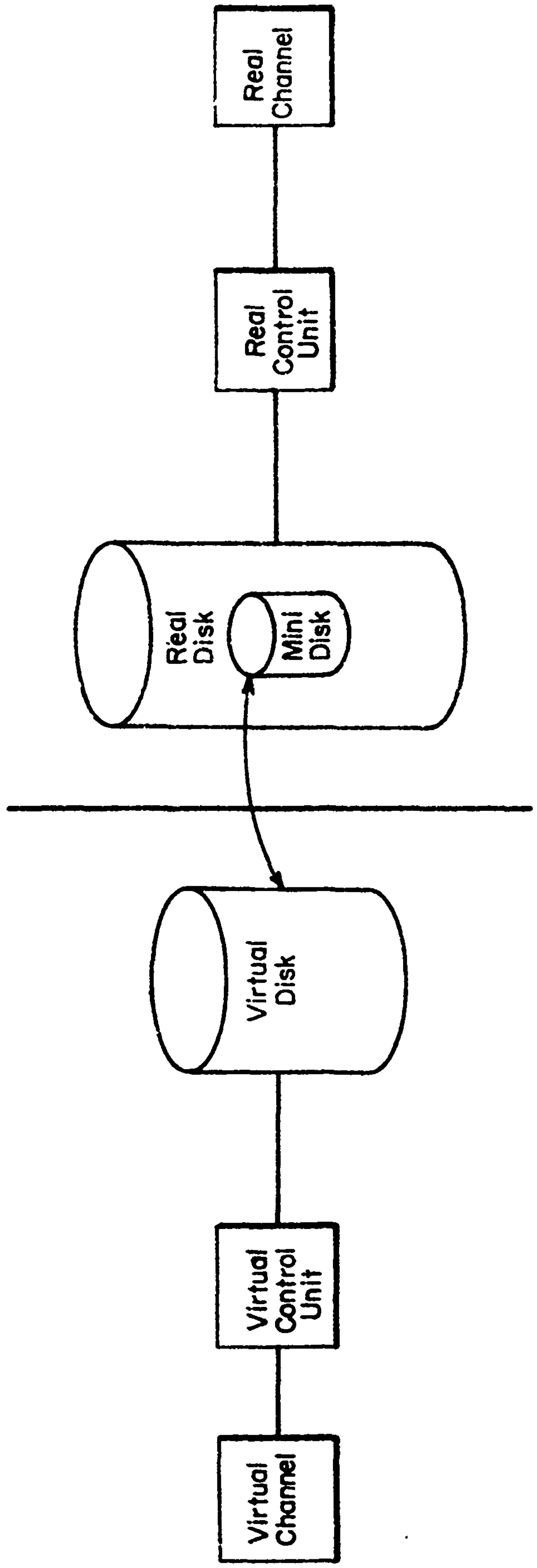
Spooling is the third method which CP-67 uses to simulate I/O devices. Each virtual machine has attached to it, a virtual card reader, virtual card punch, and virtual line printer. Rather than provide a separate real device for each virtual machine, CP-67 utilizes buffered disk files to simulate these devices. On output, data directed to the virtual printer or punch are written on a special disk file. When the real physical device becomes available, the temporary file is processed all in one piece. Similarly, card input is first read in by CP-67 and written on a disk file. Card images are then provided on each virtual I/O read request.

Dedicated I/O device simulation is the most general approach used by CP-67 and may be used for devices which are unsupported, or supported in other ways. Thus, alien devices, such as the IBM

2250 graphics console [83] may be connected providing that the virtual machine contains the appropriate I/O handling support. CP-67 need only provide very simple interrupt handling. Devices, such as the reader, punch, or printer, which are normally spooled devices, may be attached to a single virtual machine if desired. Devices, such as disk packs or telecommunications control units, which are normally supported by CP-67 through sharing, may be dedicated to a single virtual machine on a whole device basis. Thus, CP-67's handling of I/O devices is quite flexible and allows for configurations which are somewhat different from the actual resources available.

Since a virtual device address may differ from its corresponding physical device address (that is used to realize it) it is necessary to perform some kind of device mapping before an I/O transfer may take place. The device mapping is illustrated in Figure A-2. A virtual device is located on some particular virtual channel and virtual control unit. Once the device correspondence is made with the real I/O device, its particular real control unit and channel may be identified. I/O transmission requests, CCW (Channel Command Word) chains, must be translated to reflect the corresponding real I/O device involved (as well as the physical location in core memory). The CCW translation is done and the I/O request is placed on the queue for the real channel. When a completion interrupt comes in on the real channel, this information is posted back to the corresponding virtual channel.

Because of the necessity of translating CCW chains before



VIRTUAL - REAL
I/O INTERFACE
FIGURE A-2

I/O initiation, CP-67 introduces the additional restriction that no channel program is changed by the CPU or the channel during the interval between execution of the START I/O instruction and the channel end interrupt except that performed by the OS indexed sequential access method (ISAM). The restriction arises because CP-67, having translated the channel program, will have no way of knowing that it has been changed and should be translated again [8].

The actual mechanism that allows CP-67 to gain control before a virtual machine can start up I/O will be explained below in the section on the virtual processor.

A.3 CP-67's Virtual Processor

CP-67 simulates a virtual processor, e.g. 360, by using the virtual machine mechanism that was suggested in Chapter 1 and Chapter 2. Since the virtual processor and real host are similar (or identical), the virtual processor's opcodes may be executed directly on the host machine. Thus, an instruction-by-instruction interpretation is not required. However, it is necessary to prevent certain instructions from executing directly on the host machine. These instructions if executed directly by a virtual machine can cause serious difficulties in interpretation or control. Any instruction that will not be permitted to execute directly on the host machine is termed a sensitive instruction.

The approach that CP-67 adopts to prohibit the direct execution of sensitive instructions is to run the virtual machine always in (physical) problem state on the host machine. Recall that a real 360 has two modes of operation: (1) supervisor state, in which all instructions are permitted to execute, and (2) problem state, in which only a subset of the instructions are allowed direct execution. The attempt to execute a privileged instruction (one which may only be executed in supervisor state) while in problem state causes a program interrupt, i.e. trap. By running a virtual machine only in problem state, CP-67 effectively prevents the direct execution of all privileged instructions. If the sensitive instructions are privileged then CP-67 has succeeded in preventing their direct execution.

CP-67 maintains a number of tables in its supervisory area. Among them are copies of all working registers that the virtual machine believes that it has. The registers include a virtual program status word PSW. This register indicates the state that the virtual machine believes that it is in. If the virtual machine is in (virtual) problem state and a privileged instruction is attempted, CP-67 reflects this fact back to the virtual machine. The virtual machine is put into (virtual) supervisor state so that it may process the program interrupt itself. The real host machine still remains in problem state while the virtual machine is running. If the virtual machine is in (virtual) supervisor state and a privileged instruction is attempted, CP-67 recognizes that this would be a valid instruction on the real machine and simulates the effect of the instruction.

Some of the privileged instructions of the 360 which CP-67 handles in this manner are the state changing instructions Load Program Status Word (LPSW) and Set System Mask (SSM), the storage protection instructions Insert Storage Key (ISK) and Set Storage Key (SSK), the I/O instructions Start I/O (SIO), Halt I/O (HIO), Test I/O (TIO), and Test Channel (TCH), some special instructions Diagnose (DIAG), Read Direct (RDD), and Write Direct (WRD), and the special 360/67 instructions (for relocation) Load Multiple Control (LMC), Store Multiple Control (STMC), and Load Real Address (LRA). Since the Start I/O (SIO) instruction is privileged, the attempt, by a virtual machine, to perform I/O is immediately recognized. It is at this point that CP-67 may perform the mapping between virtual and real I/O devices (as described above) and then initiate the corresponding I/O operation itself. The storage protection instructions are privileged and this allows CP-67 to keep a copy of a virtual machine's protection keys and use them when the virtual machine is dispatched. Likewise, the instructions which start up the relocation system are privileged and so virtual relocation may be mapped like virtual I/O [54].

Thus, CP-67 simulates a virtual 360 processor through a combination of hardware and software. It insulates the (real) host machine from actions of the virtual machine by always running the virtual machine in problem state and in relocate mode. Because the host is in problem state, it is immune to tampering by sensitive instructions. Because the host is in relocate mode, certain reserved registers in low physical core are isolated from the virtual machine. The actions of the clock and these interrupt locations are simulated for the virtual machines by CP-67.

APPENDIX B

CASE STUDIES OF SOME III GENERATION MACHINES

B.0 Introduction

In this appendix, we provide a number of examples of third generation machines which are and which are not virtualizable. For the machines which are not, we indicate which of the empirical rules of Section 3.2 are violated. These results are summarized in Figure 3-2. This set of examples is not meant to be an exhaustive survey. Rather, it is intended to give the reader representative examples of problems that one encounters with contemporary machines. Indeed, it appears that whether or not a particular host supports virtual machines is merely a matter of chance. This is due to the fact that none of the machines cited in this section were ever designed with the thought of supporting virtual machines.

B.1 IBM 360/67

CP-67 has been in existence for several years now, so there can be little doubt concerning the 360/67's ability to support virtual machines. The initial CP-67 ran on a 360/67 but produced only virtual standard 360's, e.g. 360/65. Under these conditions, CP-67 was not self-virtualizing and so not strictly under consideration here. The extension of CP-67 to support a virtual 360/67 makes it self-virtualizing and appropriate for study here.

CP-67 is a Type I VCS which is a timesharing system, i.e. VMTSS. It produces virtual machines using the basic mechanism described in Appendix A. The 360 has two hardware modes of operation and so it is appropriate to apply the rules developed in section 3.2. Rule 1 is satisfied since the instruction execution is identical for non-privileged instructions in both supervisor and problem state. This is a characteristic of all 360's, not just the 360/67. Rule 2 is satisfied since the 360/67 DAT relocation system may be used to isolate the virtual machine from the VMM. Rule 3 is satisfied since sensitive instructions cause recoverable traps. In particular, in the 360/67, all of the instructions which can cause problems are privileged. Thus, Rule 3a is satisfied by LPSW, SSM, SVC, DIAG, LMC, STMC, LRA [as explained in Appendix A]. Rule 3b is satisfied because the relocation system makes the hardware's reserved core locations inaccessible to the virtual machine. Again, the instructions which reference the reserved registers, such as the PSW or Control Registers are privileged. Rule 3c is satisfied because the appropriate sensitive instructions are privileged. Furthermore, the use of the memory relocation system in "managing" the virtual machine does not introduce additional unsolvable complications. Thus, relocation of virtual memory may occur without error. Any traps that occur in reconciling virtual addresses with real addresses or even relocated virtual addresses with real addresses do not cause unrecoverable errors.

Two points are of considerable interest in this discussion. First, the 360/67 prechecks instructions which may cross page boundaries. Thus, instructions which may cause difficulty, such as the Add Decimal instruction, are suppressed before execution rather than terminated during it [54]. The second point is that, in reconciling relocated virtual addresses with real addresses, the DAT mechanism may be set to trap until this memory mapping is correctly established. Rule 3d is satisfied because the various instructions which reference I/O are privileged.

Thus, not surprisingly, the 360/67 satisfies all of the empirical hardware requirements stated in Section 3.2.

3.2 IBM 360/65

In this section we discuss the problem of implementing a CP-65 system. CP-65 is a VCS which runs on a 360/65 to produce virtual 360/65's.

As was mentioned above, a standard 360, e.g. 360/65, satisfies VCS hardware requirements one and two. With regard to Rule 2, the 360/65 has no relocation system so that it must rely on its storage protection system to protect the VMM. It also must use the protection system to prevent instructions from referencing the special system-wide registers (Rule 3b) stored in low core since, in general, they may be accessed without privileged instructions. Rules 3a, c, d are satisfied because the relevant instructions are privileged. Thus it is sufficient to see if the storage protection system is adequate to satisfy conditions 2 and 3b.

Since we are using the protection system to protect queries as well as alterations, we are led to the conclusion that we need the 360's five bit key store-and-fetch protect system. With this feature each 2K byte half-page is assigned a storage key via the privileged SSK instruction. Four of the bits form a store protect key and the fifth bit may be set to indicate fetch protect as well. If the four bit key in the PSW corresponds to the four bit key assigned to a 2K block, then the task is permitted to fetch or store into the block. If the keys do not match then the task has only fetch access to the block unless the fetch protect bit is also set. If the key in the PSW is zero then the task has fetch and store access to any block of storage regardless of the key stored with it.

Thus, for CP-67, a zero storage protection key in the PSW is analogous to being in supervisor state. It gives the holder more power than he may be able to use wisely. It is too much power to give to a virtual machine and so the virtual machine's key in the PSW must be some other value. Only the supervisor may actually run with storage key zero. The virtual machine runs with storage key "virtual zero" in which the supervisor performs a key translation for the virtual machine.

While this scheme adequately limits virtual memory and protects the supervisor, the dedicated locations in low core (the first 2K block) cause it trouble. Since there is no relocation scheme on a standard 360 and the machine can generate and store absolute addresses, the virtual machine runs unrelocated. Thus, the virtual machine's addresses 0-2K correspond to the real addresses 0-2K. In order to prevent the virtual machine from accessing the interval timer, new PSW's etc., the first 2K block must be fetch and store protected from all virtual machines. In order to preserve the notion of a virtual machine there must be a set of corresponding virtual special registers stored somewhere else. These may be kept in upper core. Attempts to access registers will be trapped by the storage protection mechanism and the supervisor may simulate the effect. This method of using the protection keys to prevent the virtual machine from executing sensitive instructions works. Unfortunately since the protection system on the 360 involves 2K blocks of storage, the core above byte location 128 (decimal) in block zero is protected unnecessarily. As a result any instruction in core location less than 2K or any instruction accessing data in core location less than 2K, will be trapped automatically. This will occur for all instructions, not just for the sensitive ones. Consequently in order to preserve the virtual machine notion, the supervisor must simulate each such instruction. This can cause a significant loss in efficiency.

Actually, there is at least one somewhat unlikely condition under which such an instruction may not be correctly simulated. If a storage-storage type instruction, like ADD DECIMAL is interrupted <u>during</u> execution, due to a protection exception, the instruction is terminated and partial results may be lost.

3.3 HITAC 8400

The HITAC 8400 is the Japanese equivalent of the RCA Spectra 70/45. As such, it is very similar to a standard 36 , e.g. 360/65, and so, the preceding discussion is completely applicable. It satisfies the empirical hardware requirements in the same way that the 360/65 did, above.

3.4 IBM 360/85

The IBM 360/85 is a standard member of the 360 family. Its interior decor is similar to the other standard models, e.g. 360/65. Its principal structural change from the other models is the introduction of a small buffer memory called the cache. Since the cache is invisible at the machine architecture level, i.e. not part of the interior decor, it need not be virtualized, and need not even enter into virtual machine considerations. Since any standard 360 may be virtualized, the 360/85 may be.

3.5 Data General NOVA

The Data General NOVA and SUPERNOVA are sixteen bit minicomputers. Recent work by the author and L. Dickman has led to the design of a VCS for the NOVA. The virtual NOVA is facilitated by the memory allocation and protection option that is available. This option introduces paging and two processor modes-- supervisor and user. Since all instructions which manipulate state information, e.g. the page table registers, are "I/O instructions" and all I/O instructions are privileged, Rules 3a,b,c and d are satisfied. Rules 1 and 2 are similarly satisfied. Therefore the NOVA is virtualizable.

3.6 Honeywell DDP 516

The Honeywell DDP-516 is a 16 bit mini-computer with two hardware modes of operation, a store (but not fetch) protection system, and no relocation system. The machine cannot be virtualized for a number of different reasons.

Rule 3 is generally violated. Some of the sensitive instructions are trapped, either as a result of the privilege mechanism or the protection mechanism. However, for those instructions, the trap is deferred one full instruction execution. Thus, the state of the machine will be changed before the supervisor gains control. In fact, it is even possible for the instruction following the sensitive instruction to store into that instruction before the trap has occurred. Now, the VMM has no hope of simulating the instruction.

Rule 3a is violated in a manner similar to the PDP-10. [See below.] The instruction used to return from supervisor state to problem state is ERM (Enter Restricted Mode). Unfortunately, this instruction is not privileged, Thus, its execution in virtual supervisor state will not cause a trap. A virtual state change may occur without the knowledge of the VMM.

Rule 3b is violated because of the lack of a relocation or fetch protect system. This difficulty by itself need not be fatal since the introduction of a suitable restriction can still produce a useful virtual machine (as in Fuchi [49]). However, Rule 3b is violated in another more severe manner with the IMA (Interchange Memory and Accumulator) instruction. If IMA is attempted into an area that is protected, the accumulator will be loaded before the protection violation is signalled. Consequently, it is impossible to recover the original contents of the accumulator and simulation is not possible.

3.7 GE 635, 655 (Honeywell 6000)

The GE 635 cannot be virtualized because of violations of Rules 3a and 3c. The LDT (Load Timer) and LBAR (Load Base Address Register) instructions are sensitive instructions that null execute rather than trap when issued in problem state (slave mode). The GE 655, a successor to the 635 attempts to correct this error by providing a trap on LDT or LBAR in problem state. However, there are still difficulties with the machine because of visibility to the physical master/slave mode via the non-privileged store indicators instruction.

3.8 Multidata Model A (SEL)

The Multidata Model A is a 16 bit mini-computer with a hardware implemented paging system. A virtual machine system is impossible since Rule 3 is completely violated. All privileged instructions execute as NOP when in problem state. These instructions include the status switching instructions and I/O instructions.

3.9 XDS 940

The XDS 940 cannot be virtualized because of a fundamental violation of Rule 1. Bit 0 of each instruction word indicates whether or not the memory mapping is to be on for that instruction. The bit has effect only in supervisor state but is ignored in problem state where it is assumed that all addresses are mapped. Thus, running the virtual supervisor state as physical problem state does not provide a mechanism to distinguish between mapped and unmapped addresses.

3.10 DEC PDP-10

The DEC PDP-10 may not be virtualized because it violates Rule 3a. The JRSTF (Jump and Restore Flags) or "JRST 1," (Jump to User Program) instructions are used to return from supervisor state to problem state. When executed in (physical) problem state, the instructions do not trap. Thus, it is impossible to detect a virtual state change.

Another violation of Rule 3a concerns the accessibility of the "flags" which indicate what physical mode the machine is in. Thus, a program in virtual supervisor state might discover from the flags that it is in (physical) problem state and become "confused".

As with several other machines which violate Rules 1 or 3a, it is possible to construct a hybrid virtual machine on the PDP-1 . [See Section 3.3.] The HVM is possible since an attempt by a virtual machine to enter supervisor state traps to the VMM. The means of entering supervisor state are via a (privileged) JUO. Thus, an HVM, Type I or Type II, is possible on the PDP-10. The organization of a Type II HVM is discussed in Appendix C.

3.11 BBN TENEX

TENEX [2_] is a PDP-10 that has been modified to include hardware paging. This modification has not changed the situation with respect to the JRSTF instruction. Thus, TENEX is not virtualizable but an HVM may be constructed.

APPENDIX C

CASE STUDIES OF THIRD GENERATION TYPE II SYSTEMS

C.0 Introduction

In this appendix, we consider examples of several different operating systems which do and do not support Type II (extended machine host) VMM's. For the machines which are Type II virtualizable, we indicate the modifications that had to be made to support the empirical software requirements. For the systems which are not, we indicate the empirical rules which are violated. These results are summarized in Figure 3-8. The PDP-10 did not satisfy the hardware requirements but the ITS operating system satisfies the software rules. Consequently, we are able to exhibit a Type II hybrid virtual machine for this system. As with the hardware examples of Appendix B, the operating systems investigated in this section were not designed with the thought of supporting virtual machines.

C.1 HITAC 8400

The HITAC 8400 [49] has a Type II virtual machine which runs under the TCS/TDOS operating system. The HITAC 8400 is (more or less) Type I virtualizable [see Appendix B]. TOS/TDCS satisfies software rule S1 but does not satisfy S2 or S3. Consequently, modifications were made to provide protection on an ad hoc basis. Other ad hoc solutions were introduced to eliminate some of the violations of Rule S3. For example:

> A supervisor call instruction (SVC) is not a privileged instruction but it causes interrupts changing the current program state to P3. As TOS/TDOS had no set-exit function for this instruction, we used an illegal code instruction instead of a SVC instruction to permit simulation through a (sic) illegal code trap [49].

The authors indicate some of the Type II requirements.

> The supporting operating system must provide a "set exit" or "ON" function for these types of traps caused by hardware. Aside from privileged instructions, the so-called SVC (supervisor call) instruction must also be "set exited" and simulated. The SVC normally communicates with the current supervisor but now it must be forced to direct to the supervisor simulated [49].

Recognizing the inelegance of their system, the authors propose some improvements:

> As for the TOD/TDOS, the interval timer should have finer precision and set-exit functions for address errors and SVC should be provided as standard features. In addition, "set storage key," "set time," and "set CAW" functions should be prepared [49].

C.2 360/67 UMMPS

A Type II virtual machine has been constructed on the UMMPS system for the IBM 360/67 [4,60]. Since UMMPS did not satisfy Rules S2 and S3, it was modified to include several new primitives. The most important of these is called SVC SWPTRA. A very brief description of the virtual machine monitor (called "the monitor", below) and the action of SVC SWPTRA is given in an informal user's guide.

> When you run the monitor, you tell it (by way of the VARY command) which virtual devices are attached to which real devices. You then issue the IPL command and the monitor fakes the IPL sequence. After this, you

> communicate with the virtual system by way of commands which make available the 360 console functions (display and alter memory, PSW restart, external interrupt, console request, etc.)
>
> The monitor acquires space for and loads the virtual system into segment 3. The reason for using a new segment is this: when the virtual machine is running, it must have addresses 0-256K (or 0-512K or whatever). This is accomplished by SVC SWPTRA, which swaps the segment table entries for segment 0 and segment 3, detaches segments 0 and 1, and 4+, and transfers into the virtual system. This takes care of the addressing problem.
>
> Now the virtual machine is running in problem state. Therefore, whenever it attempts a privileged operation (SIO, LPSW, SSK, etc.), a program interrupt occurs. At this point, the supervisor (UMMPS) swaps segments 0 and 3 back again and reconnects the other segments and transfers into the monitor. At this point the monitor can fake the privileged operation or not as it sees fit.
>
> The sequence is therefore like this: The monitor transfers control to the virtual machine by way of an SVC SWPTRA. Any task interrupt in the virtual machine (program interrupt, time overflow, asynchronous device exit, etc.) will cause the supervisor to re-establish standard addressing and transfer into the monitor. [60]

Thus, SVC SWPTRA comes fairly close to being a primitive that starts up a subprocess. In its case, the particular choice of which interrupts trap back to the virtual machine supervisor have been decided in advance. There is no general flexibility in deciding more precisely what is permitted in the subprocess. However, since the subprocess is a particular kind of virtual machine, e.g. System/360 (360/65), there is no real need to provide the additional flexibility.

Note that the UMMPS virtual machine is not self-virtualizing. That is, the VMM runs on a 360/67 to produce a virtual 360/65. The work is currently being extended to include a virtual 360/67.

C.3 OS/360

OS/360 [66,84] is the principal operating system being run with the IBM System/360. As such it is extremely difficult (and possibly foolish) to implement ad hoc changes in it. One of the principal reasons that OS/360 is run is so that an installation will be somewhat compatible with the rest of the IBM world. Making ad hoc changes to OS/360 sabotages that objective.

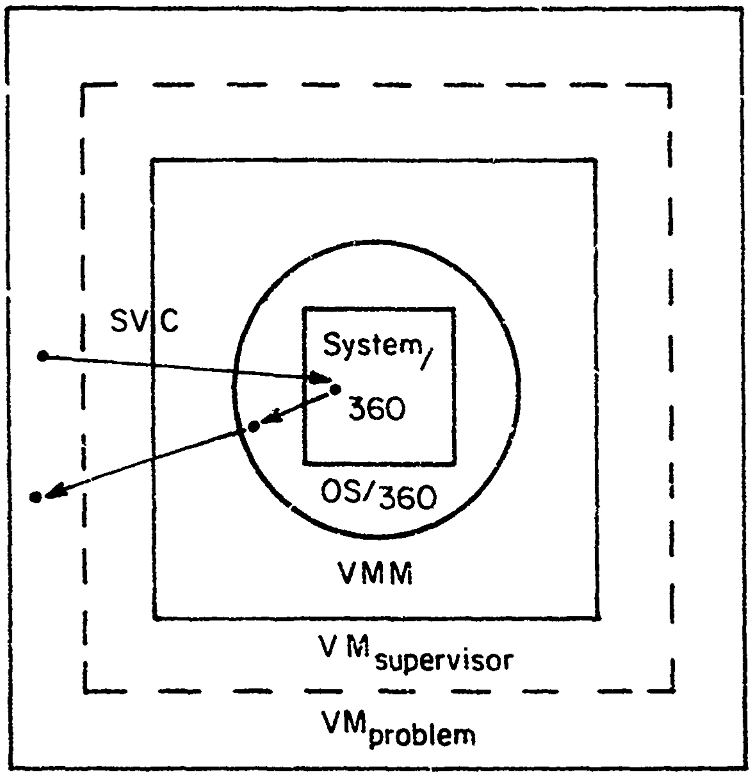
Thus, in contrast to UMMPS, say, where the operating system was modified to support virtual machines, one would not expect to find changes to OS/360. This, in part accounts for the lack of Type II virtual machines under OS/360 since a standard 360 is Type I virtualizable. [See Appendix B.]

In this section, we will examine some aspects of OS/360 and show how the public version of OS (pre-1970) violates Rules S2 and S3. We use as a guide, "Notes on Construction of Subsystems Within Operating System/360" by E. Satterthwaite [100]. Satterthwaite's objective is slightly different from implementing virtual machines, but the difficulties he uncovers are equally applicable. Since OS/360 is continually evolving, remarks made about it often tend to be time-dependent.

OS/360 violates rule S2 because all memory available to a job is assigned a single key, and that key appears in the PSW of every task associated with that job. This implies that the virtual machine is able to destroy the VMM. Satterthwaite suggests that OS/360 provide a pair of SVC instructions, which, after appropriate validity checking, switch the protection key of a part of a partition (region) between zero and the key associated with the corresponding job.

OS/360 violates rule S3 in many different ways. Of the interrupts and exceptions, only program interrupts may be signalled to the virtual machine monitor in such a way as to be recoverable. For program interrupts in the virtual machine, e.g. attempts to execute privileged instructions, the VMM is able to issue an SVC SPIE to give the address of the appropriate "error handling" routine.

Other interrupts such as I/O, timing or SVC interruptions are not handled as well. For example, although the virtual machine supervisor can specify a time interval and an exit routine to be given control upon the expiration of that interval, there does not appear to be a method by which the exit routine may examine or alter the PSW or general register contents existing at the point of interruption. Thus, it is impossible to simulate the effect of this instruction. The worst failing of OS/360 concerns the lack of an hierarchical viewpoint concerning SVC's. There does not appear to be a way in OS for a virtual machine supervisor to mask out the SVC's of its virtual machine. Such a mask should enable a specified exit routine to conditionally inhibit the execution of selected supervisor services. Without this trap, there does not appear to be a way for the virtual machine's SVC's to be signalled, a violation of Rule S3a. Thus, in Figure C-1, execution of an SVC by the virtual machine (in problem state) causes a hardware trap to the



OS/360 DOES NOT MASK SVC'S

FIGURE C-1

system and control passes to OS/360. CS/360 then interprets the SVC as a request for its service, performs the service and returns to the calling virtual machine (in problem state). Thus the VMM never gets control and it cannot put the virtual machine into virtual supervisor state for the intended SVC handling. Note the similarity between the need to mask out SVC's of the extended machine and the need to cause traps on sensitive instructions in the hardware machine. A masking SVC is, thus, very similar to the TCO instruction of Section 3.4.

C.4 MIT PDP-10 ITS

While the PDP-10 is not virtualizable, it is, however, possible to construct an HVM on it. [See Section 3.3 and Appendix B.] Furthermore, when running under MIT's ITS, Incompatible Timesharing System [42,43], it becomes feasible to construct a Type II Hybrid Virtual Machine (HVM) [56].

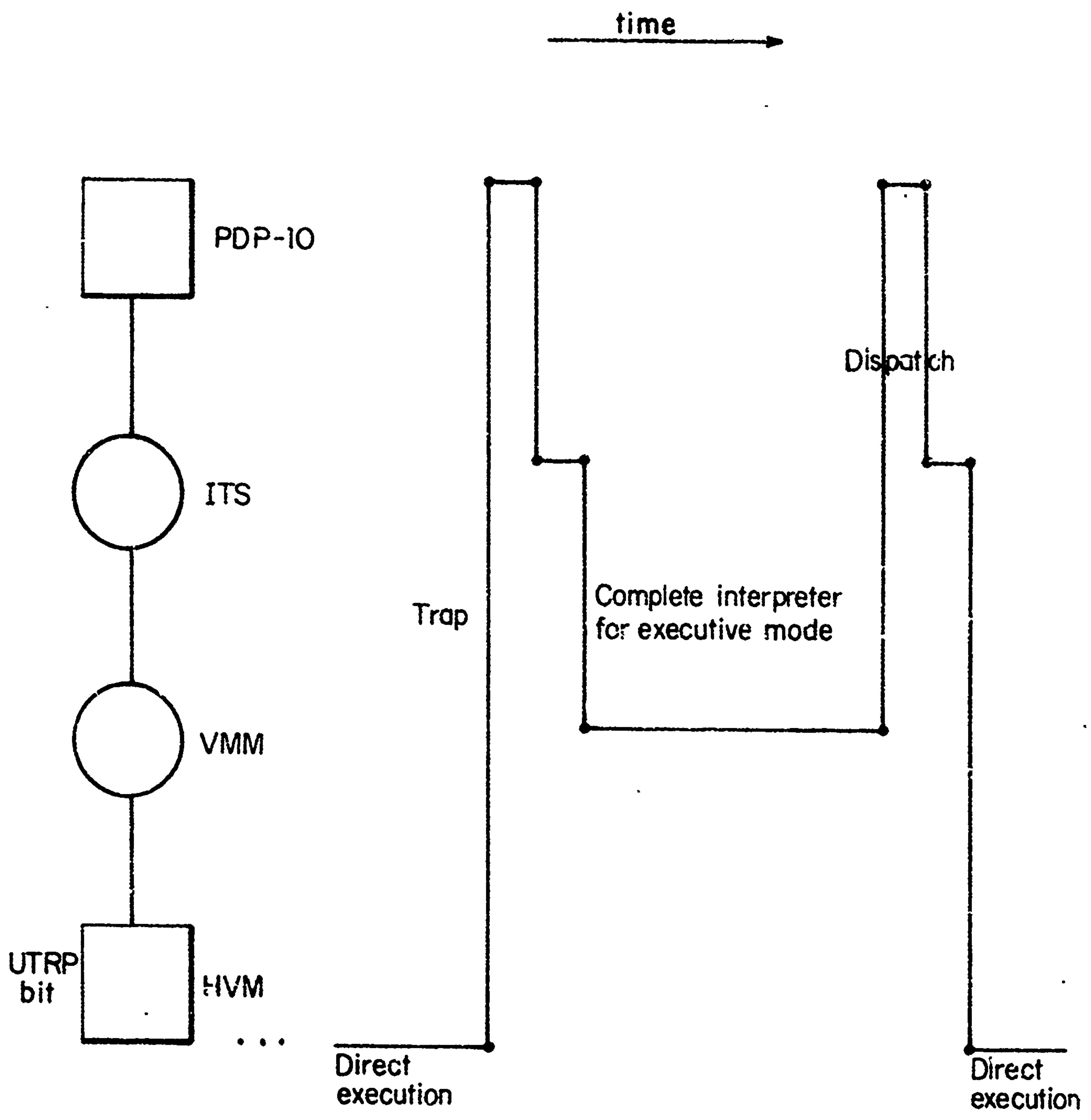
ITS is an hierarchical operating system in which one process can control a sub-tree of other processes. In particular, ITS provides primitives (UUC's) which enable a process to create a subprocess, read the core image of the subprocess, and get alerted when the subprocess attempts to reference outside its core image or attempts to execute a privileged instruction or UUO. The entire signalling mechanism (user trap mode) is controlled by one bit, called the UTRP bit, in the process's control block ("system variables"). A UUO to turn the UTRP bit on or off is available to the parent process.

Thus, it becomes a simple matter to design a Type II HVM.

The VMM is the superior, the virtual machine the inferior in a process tree. The VMM sets the UTRP bit to direct all inferior traps to the VMM. The VMM permits the HVM to execute directly in user mode. When the HVM attempts to enter virtual executive mode, the VMM is alerted. The VMM interprets each instruction until the virtual machine returns to user mode. See Figure C-2.

C.5 CMS, The Cambridge Monitor System

CMS [85] is a rather bare single-user conversational operating system which runs on a standard 360. It is most often found in conjunction with CP-67. In this usage, CMS runs on a virtual machine under CP-67. CMS provides a number of operating system primitives that may be used by a prospective VMM. However, since CMS is a single user conversational system (most often run on a virtual machine), it does not provide any features that may be used to protect or signal the VMM as required by rules S2 and S3. At the same time, CMS does not make the hardware features (instructions) to accomplish these tasks inaccessible. Thus, with rather simple alterations, the necessary mechanisms may be added to CMS. While no virtual machine under CMS has been discussed in the literature, various installations, such as MIT Lincoln Laboratory, have made most of the modifications that make this possible.



TYPE II HVM UNDER PDP-10 ITS

FIGURE C-2

APPENDIX U

GLOSSARY

The intent of the glossary is to provide, in one place, very brief summary definitions for some of the terms or abbreviations used in the thesis. For this reason there is no attempt made to give very complete or particularly formal definitions. This glossary is merely intended as a convenience for the reader, reminding him of certain terms or abbreviations. For more information, the reader is directed to the section numbers indicated with most entries.

architecture: attributes of a system as seen by the programmer, i.e. the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation

associator: hardware device for performing rapid parallel search

control program: virtual machine monitor

EMM: extended machine monitor mapping program between two different operating systems [2.1]

emulation: technique for simulating an interior decor on a machine with some different one

exception: trap caused by process violation [different from fault]: control should remain within executing virtual machine [4.1]

family: compatible series of computers with possibly minor interior decor differences, e.g. 360/40 and 360/67,

family-virtualizing: the virtual machine is not identical to the host but is a member of the same computer family [2.1] firmware: microprogram which defines interior decor

FV: family-virtualizing [2.1]

f: virtual machine map [4.2]

ϕ: process map [4.1]

generation: architectural characteristics of the principal machines of that generation [In the thesis, generation always refers to architecture, never to physical implementation, etc.]

HV: hardware virtualizer [4.5]

HVM: hybrid virtual machine [3.3]

hardware virtualizer: collective title for hardware-firmware design which directly supports the (IV generation) VCS model [4.5]

host machine: bare machine on which virtual machine monitor runs [2.1]

host operating system: operating system under which Type II virtual machine monitor runs [2.1]

hybrid virtual machine: virtual machine in which all instructions issued within the most privileged layer are software interpreted and all other instructions execute directly [3.3]

II generation: second generation architectures typical of IBM 7090 [4.3]

III generation: third generation architectures typical of IBM 360 [3.1]

IV generation: fourth generation architectures with elaborate, rich process structure implemented in firmware [4.1]

interior decor: software-visible definition of a system, i.e. architecture

intra-level map: map within a virtual machine level which establishes the (layer) structure within that level; the map may be visible to privileged software of that level [4.1,4.2]

inter-level map: (VMmap) map between two levels of virtual machines, e.g. level n to level n+1, which is invisible to the higher level, e.g. level n+1 [4.2]

invisible: unable to be detected in software layer: restricted access structure (within a level) such as master/slave modes or rings [2.1,4.2]

level: number of VMmaps which must be successively applied to map a virtual resource name into a real resource name: also corresponds to the number of syllables in the VMID: thus, the real machine is level 0

native mode: operation of system, e.g. CPU, with preferred order code, not emulation

order code: instruction set

PCB: process control block, process status kept in system base [4.1]

P-name: name used by process to refer to some entity

process map: abstraction of the map from process names, e.g. segment numbers, to resource names, e.g. memory locations [4.1]

privileged instruction: instruction which executes (correctly) only in privileged mode, i.e. supervisor state, ring zero [3.1]

R-name: real resource name [4.2]

R-B: relocation-bound form of address relocation, as in DEC PDP-10

reloadable control store: modifiable microprocessor storage

resource map: virtual machine map

ring: layered protection system, generalization of master/slave mode, supervisor/problem state [4.1]

ring 0: most privileged ring [4.1]

ROOT: origin pointer of system base of typical IV generation machine [4.1]

RPW: running process word, in system base, contains identifier of active process [4.6]

SV: self-virtualizing [2.1]

self-virtualizing: the processor of the virtual computer system is identical to the host [2.1]

semaphore: means for interprocess communication in typical IV generation computer system [4.1,4.6] sensitive instruction: instruction which, because of III generation virtual machine software construction will give incorrect result if permitted to execute directly on the host machine [3.2]

supervisor call: (SVC) instruction which is used to communicate with the operating system [2.1]

system base: database directly accessed by firmware (and updated by privileged software) of typical IV generation system to support implementation of the process model [4.1]

TOC: trap on opcode instruction, ad hoc suggestion for designing a (III generation) machine in which privileged instructions are chosen at execution time, and thus can guarantee III generation software virtual machine construction [3.4]

Type I VMM: the VMM runs on a bare machine host [2.1]

Type II VMM: the VMM runs on an extended machine host, under the host operating system

UUO: supervisor call instruction on DEC PDP-10

VCS: virtual computer system

VCSCB: VCS control block in system base of hardware virtualizer provides VMmap for virtual machine [4.5]

VCSTAB: VCS table in system base of hardware virtualizer contains pointers to VCSCB's for virtual machines [4.5]

VM: virtual machine [never virtual memory]

VMCB: same as VCSCB [4.5]

VMID: virtual machine identifier register in hardware virtualizer indicates active virtual machine [4.5]

LVMID: load VMID instruction in hardware virtualizer, appends next syllable to the VMID register and dispatches the virtual machine [4.5]

VMM: virtual machine monitor, software which mediates between the virtual machine and host [2.1]

VMTSS: virtual machine timesharing machine [2.2]

VP: Venice paper or proposal: proposal for firmware assisted virtualizer made in [51] [4.4]

V-name: virtual resource name [4.2]

VCS model: composition of abstract process map and resource map to express running a process on a virtual computer system [4.2]

virtualization: act of creating/running virtual machine(s)

virtualizable: able to create (self-virtualizing) virtual machine(s)

VM-fault: resource fault by virtual machine to VMM [4.2]

VM-level fault: VM-fault

VMmap: abstraction of map between two levels of virtual resource allocation [completely distinct from process map] [4.2]

VM-recursion property: in self-virtualizing virtual machine, ability to run a VMM on the VM [2.1.4.2]

BIBLIOGRAPHY

1 Ackerman, W.B. and Plummer, W.W. An implementation of a multiprocessing computer system. ACM Symp. on Operating System Principles, Gatlinburg, Tennessee, (Oct.,1967).

2 Adair, R. and Bard, Y. CP-67 measurement method. IBM Corp., Cambridge Scientific Center Rept. No. G320-2072 (May,1971).

3 Adair, R., Bayles, R.U., Comeau, L.W. and Creasy, R.J. A virtual machine system for the 360/40. Cambridge Scientific Center Rept No. G320-2007 (May, 1966).

4 Alexander, M.T. Time sharing supervisor programs. Univ. of Michigan Comp. Center, (May, 1969, revised May, 1970).

5 Amdahl, G.M. and Amdahl, L.D. Fourth-generation hardware. Datamation (Jan., 1967).

6 Amdahl, G.M., Blaauw, G.A. and Brooks, F.P. Architecture of the IBM system/360. IBM Journal of Research and Development (April, 1964).

7 Amdahl, L.D. Architectural questions of the seventies. Datamation (Jan., 1970).

8 Ancilotti, P., Cavina, R. and Lijtmaer, N. Virtual input-output in a virtual environment. ACM International Comp. Symp. Proc., Venice, Italy, (April 12-14, 1972), pp. 302-312.

9 Auroux, A. and Hans, C. Le concept de machines virtuelles. Revue Francaise d'Informatique et de Recherche Operationelle, 2e annee, 15 (1968), pp. 45-51.

10 Bairstow, J.N. Many from one: the virtual machine arrives. Computer Decisions (Jan., 1970), pp. 29-31.

11 Balzer, R.M. The ISPL language specifications. Rand Corp., R-563-ARPA (Aug., 1970).

12 Balzer, R.M. ISPL machine principles of operation. Rand Corp., R-562-ARPA (Aug., 1970).

13 Bard, Y., Margolin, B., Peterson, T. and Schatzoff, M. CP-67 measurement and analysis, I: regression studies. IBM Corp., Cambridge Scientific Center Rept. G320-2061 (June, 1970).

14 Bord, Y. Performance criteria and measurement for a time-sharing system. *IBM Sys. J.* 10, 3 (1971), pp.193-216.

15 Baskin, H.B., Borgerson, B.R. an Roberts, R. PRIME--A modular architecture for terminal-oriented systems. *Proc. of SJCC* (1972), pp.431-437.

16 Bensoussan, A., Clingen, C.T. and Daley, R.C. The multics virtual memory. *2nd ACM Symp. Op Sys. Prin.* , Princeton U. (Oct. 20-22, 1969), pp.30-42.

17 Bell, C.G. and Newell, A. *Computer structures: readings and examples*. McGraw Hill, N.Y. (1971).

18 Belliro, J. and Potin, Ph. Mecanismes d'un hyperviseur. *Report of IBM Scientific Center*, Grenoble, France.

19 Blaauw, G.A. Hardware requirements for the fourth generation. *Fourth generation computers: user requirements and transition*. ed. F. Gruenberger, Prentice Hall. (1970).

20 Bobrow, D.G., Burchfiel, J.D., Murphy, D.L. and Tomlinson, R.S. TENEX, a paged time-sharing system for the PDP-10. *Comm. of ACM* 15, 3 (March, 1972), pp. 135-143.

21 Brooks, F.P., Jr. Architectural philosophy. *Planning a Computer System*, ed. W. Buchholz, McGraw Hill, N.Y. (1962), pp. 5-16.

22 Buzen, J. Optimizing the degree of multiprogramming in demand paging systems. *IEEE Comput. Soc. Conf.* Boston, Mass. (Sept. 22-24, 1971). pp. 141-142.

23 Calloway, P.H. Performance considerations for the use of the virtual machine capability. IBM Corp., T.J. Watson Research Lab., Yorktown Heights, N.Y., *Rept. RC-3360* (May 12, 1971).

24 Calloway, P.H., Considine, J.P. and Thompson, C.H. Uses of virtual storage systems in a scientific environment. *IBM Systems Journal* 11, 3 (1972) pp. 200-218.

25 Cheatham, T.E. The recent evolution of programming languages. *Proc. IFIP Cong.*, North Holland Pub. Co., Amsterdam (1971).

26 Comeau, L.W. A study of the effect of user program optimization in a paging system. *ACM Symp. on Op. Sys. Prin.*, Gatlinburg, Tennessee (Oct., 1967).

27 Comeau, L.W., Lindquist, A.B. and Seeber, R.R. A time-sharing system using an associative memory. *Proc. of IEEE* 54, 12 (Dec., 1966), pp. 1774-1779.

28 Corbato, F.J. System requirements for multiple access time-shared computer. MAC-TR-3, MIT, Cambridge, Mass.

29 Corbato, F.J., Saltzer, J.H. and Clingen, C.T. Multics--the first seven years. Proc. of SJCC (1972), pp. 571-583.

30 Corbato, F.J. and Vyssotsky, V.A. Introduction and overview of the MULTICS system. AFIPS Proc., FJCC 27, I (1965), pp. 185-196.

31 COSINE Task Force VIII. An undergraduate course on operating systems principles (June, 1971).

32 CP-67/CMS, Program 360 D-05.2.005, IBM Corp., Program Information Dept., Hawthorne, N.Y. (June, 1969).

33 Creasy, R.J. General description of the research time sharing system with special emphasis on the control program. Unpublished memo No. 1 of IBM Scientific Center, Cambridge (Jan., 1965).

34 Daley, R.C. Proposed 645 follow-on processor specification. Multics 645 Follow-on Hardware Design Memo, MHDM-1., MIT, (June 23, 1970).

35 Daley, R.C. and Dennis, J.B. Virtual memory, processes, and sharing in Multics. Comm. of ACM 11, 5 (May 1968), pp. 306-312.

36 Dartmouth University. Specifications for GECOS III executive commands as implemented on the DTSS (Phase II) GECOS subsystem. Kiewitt Computer Center, Dartmouth University, Hanover, N.H.

37 Denning, P.J. Third generation computer systems. Computing Surveys 3, 4 (Dec., 1971).

38 Denning, P.J. Virtual memory. Computing Surveys 2, 3 (Sept., 1970), pp. 153-189.

39 Dennis, J.B. Segmentation and the design of multiprogrammed computer systems. JACM 12, 4 (Oct., 1965), pp. 589-602.

40 Dennis, J.B. and Van Horn, E.C. Programming semantics for multiprogrammed computation. CACM 9, 3 (March, 1966), pp. 143-155.

41 Dijkstra, E.W. The structure of THE multiprogramming system. CACM 11, 5 (May, 1968), pp. 341-346.

42 Eastlake, D.E. ITS status report. Memo No. 238, AI Lab, MIT (April, 1972).

43 Eastlake, D.E. ITS 1.5 Reference Manual. MIT Artificial Intelligence Laboratory Memo No. 161A (also MAC-M-377). July, 1969.

44 Emulating DOS under OS for IBM System/370. IBM Systems Reference Library Form GC26-3777, IBM Corp., White Plains, N.Y.

45 England, A.C. and Byer, R. A DEC 10/50 simulator designed to run on ITS. Unpublished Project MAC memo (Aug., 1972).

46 Evans, D.C. and LeClerc, J.Y. Address mapping and the control of access in an interactive computer. AFIPS Conf. Proc. 30 (1967 SJCC), pp. 23-30.

47 Field, M.S. Multi access systems-the virtual machine approach. IBM Cambridge Scientific Center Rept. 320-2033 (Sept., 1968).

48 Forgie, J.W. A time-and memory-sharing executive program for quick response. 1965 FJCC.

49 Fuchi, K. and Kaneda, Y. Utilization of a simple paging mechanism. Bul. Electrotech Lab. 34, 4 (1970), pp. 55-63.

50 Fuchi, K., Tanaka, H., Namago, Y., and Yuba, T. A program simulator by partial interpretation. 2nd Symp. on Op. Syst. Prin., Princeton, New Jersey (Oct., 1969), pp. 97-104.

51 Gagliardi, U.O. and Goldberg, R.P. Virtualizeable architectures. Proc. of 1972 ACM International Conp. Symp., Venice, Italy (April, 1972), pp. 527-538.

52 Goldberg, R.P. Hardware requirements for virtual machine systems. HICSS-4, Hawaii International Conference on System Sciences. Honolulu, Hawaii (Jan., 1971).

53 Goldberg, R.P. Virtual machines: semantics and examples. IEEE Comput Soc. Conf. Boston, Mass. (Sept., 1971) pp. 141-142.

54 Goldberg, R.P. Virtual machine systems. MIT Lincoln Laboratory Report No. MS-2687 (also 28L-0036). Lexington, Mass. (Sept., 1969).

55 Goldberg, R.P. Unpublished lecture, Applied Mathematics 251a, Harvard University (Dec. 7, 1970).

56 Goldberg, R.P. and Galley, S.W. A Type II Hybrid Virtual Machine for the DEC PDP-10 under ITS. MIT Project MAC internal note (Oct., 1972).

57 Graham, R.M. Protection in an information processing utility. CACM 11, 5 (May 1968), pp. 365-369.

58 Hans, C. et al. GMS guide de l'utilisateur. Rept. of IBM Scientific Center of Grenoble, Grenoble, France (July, 1972).

59 Hoernes, G.E. and Hellerman An experimental 360/40 for time-sharing. Datamation 14, 4 (April, 1968), pp. 39-42.

60 Hogg, J. and Madderom, P. The virtual machine facility--how to fake a 360. Univ. of British Columbia--Univ. of Michigan Internal Note.

61 Holloway, J. PDP-10 paging device. Hardware Memo 2, MIT AI Lab. (Feb., 1970).

62 Holloway, J. Paging device signal names. Hardware Memo 2A, MIT AI Lab. (Feb., 1970).

63 Husson, S.S Microprogramming: principles and practices. Prentice-Hall, Englewood Cliffs, New Jersey (1970).

64 IBM, Adding computer virtually. IBM Computing Rept., Vol. III, 2 (March, 1967).

65 IBM, Control program-67/Cambridge monitor system. IBM Type III release No. 360D-05.2.005. IBM Corp., Program Information Department, Hawthorne, N.Y.

66 IBM, OS/360 Concepts and Facilities. GC28-6535.

67 IBM, VM/370: Planning Guide. GC20-1801-0

68 IBM, IBM system/360 model 67 functional characteristics. SRL...A27-6719.

69 IBM, IBM system/360 principles of operation. SRL...A22-6821.

70 Infotech. The IV Generation. Infotech, Maidenhead, England (in publication).

71 Keefe, D.D. Hierarchical control programs for systems evaluation. IBM Sys. J. No. 2 (1968), pp. 123-133.

72 Kelch, J.A., Seawright, L.H. An introduction to CP-67/CMS. IBM Cambridge Scientific Center Rept. 320-2032 (Sept., 1968).

73 Lampson, B.W. An overview of the CAL time-sharing system. Comp. Center, Univ. of Calif., Berkeley (Sept. 5, 1969).

74 Lampson, B.W. Dynamic protection structures. AFIPS Conf. Proc. 35 (1969 FJCC), pp. 27-38.

75 Lampson, B.W., Lichtenberger, W.W. and Pirtle, M.W. A user machine in a time-sharing system. Proc. of IEEE 54, 12 (Dec., 1966), pp. 1766-1774.

76 Lauer, H.C. and Snow, C.R. Is supervisor-state necessary? International Comp. Symp. Proc., Venice, Italy (April 12-14, 1972), pp. 293-301.

77 LeClerc, J.Y. Memory structures for interactive computers. Ph.D. Thesis, Univ. of Calif., Berkeley, Calif., Project Genie, Document 40.10.110 (May, 1966).

78 Lindquist, A.B., Seeber, R.R. and Comeau, L.W. A time-sharing system using an associative memory. Proc. IEEE 54, 12 (Dec. ,1966), pp. 1774-1779.

79 Liskov, B.H. The design of the venus operating system. CACM 15, 3 (March, 1972).

80 Madnick, S.E. Storage hierarchy systems. Ph.D. Thesis, MIT, Project MAC (May, 1972).

81 Madnick, S.E. Time-sharing systems: virtual machine concept vs. conventional approach. Modern Data 2, 3 (March, 1969), pp. 34-36.

82 Mallach, E.G. Emulation: practice and principles. ACM Computing Surveys (forthcoming).

83 McGrath, M. Virtual machine computing in an engineering environment. IBM Systems Journal 11, 2 (1972) PP. 131-149.

84 Mealy, G.H., Witt, B.I. and Clark, W.A. The functional structure of OS/360. IBM Systems Journal 5, 1 (1966) pp. 2-51.

85 Meyer, R.A., and Seawright, L.H. A virtual machine time-sharing system. IBM Systems Journal 9, 3 (1970), pp. 199-218.

86 MIT Project MAC The Multiplexed Information and Computing Service: Programmer's Manual. MIT Project MAC, Rev. 10, 1972.

87 Morris, D. The nature and benefits of modular operating systems. The IV Generation. Infotech, Maidenhead, England (in publication).

88 Morris, D. and Cotlofson, G.C. An implementation of a segmented virtual store. Dept. of Comp. Sci., University of Manchester, Manchester, England.

89 Morris, D. and Cotlofson, G.D. A virtual processor for real time operation. Dept. of Comp. Sci. University of Manchester, Manchester, England.

90 Motobayashi, S., Masuda, T. and Takahashi, N. The HITAC 5020 timesharing system. Proc. ACM Summer Conf. (1969).

91 O'Neill, R.W. Experience using a time-shared multi-programming system with dynamic address translation. AFIPS 30 (1967).

92 Organick, E.I. The multics system: an examination of its structure. MIT Press, Cambridge, Mass. (1972).

93 Parmelee, R.P. Preferred virtual machines for CP-67. IBM Cambridge Scientific Center Rept. G320-2063. (to appear).

94 Parmelee, R. Virtual machines: some unexpected applications. Proc. IEEE Comput. Soc. Conf. Boston, Mass. (Sept., 1971).

95 Parmelee, R.P, Peterson, T.I., Tillman, C.C. and Hatfield, D.J. Virtual storage and virtual machine concepts. IBM Sys. J.2 (1972), pp. 99-129.

96 Reigel, E.W., Faber, U. and Fisher, D.A. The interpreter--a microprogrammable building block system. Proc. of SJCC (1972), pp. 705-723.

97 Rosin, R.F. Contemporary concepts of microprogramming and emulation. Comp. Surv.1 (Dec., 1969), pp. 197-212.

98 Rosin, R., Frieder, G. and Eckhouse, R. An environment for research in microprogramming and emulation. 4th Workshop on Microprogramming, Santa Cruz (Sept., 1971), preprints.

99 Saltzer, J.H. Traffic control in a multiplexed computer system. MAC-TR-30 (July, 1966).

100 Satterthwaite, E. Notes on construction of subsystems within operating system/360. Stanford Univ. Comp. Group Rept. CGTM No. 43 (March, 1968).

101 Sayre, D. Adding computers virtually. IBM Corp., Comput. Rept. III, 2 (March, 1967) pp. 12-15.

102 Sayre, D. Is automatic 'folding' of programs efficient enough to displace manual? CACM 12, 12 (Dec., 1969), pp.656-660.

103 Sayre, D. On virtual systems. IBM Corp., T.J. Watson Research Lab., Yorktown Heights, N.Y. (April 15, 1966).

104 Schell, R.R. Dynamic reconfiguration in a modular computer system. MIT Project MAC, Cambridge, Mass. (June, 1971).

105 Schroeder, M.D. Performance of the GE-645 associative memory while Multics is in operation. ACM-SIGOPS Workshop on Sys. Perf. Eval., Harvard Univ. (April, 1971), pp. 227-245.

106 Schroeder, M.D. and Saltzer, J.H. A hardware architecture for implementing protection rings. CACM 15, 3 (March, 1972), pp. 157-170.

107 Schwemm, R.E. Experience gained in the development and use of TSS/360. Proc. of SJCC (1972), pp. 559-570.

108 Spier, M.J. and Organick, E.I. An operating system model featuring demand scheduling of virtual resources. Presented at 3rd Hawaii International Conf. on System Sciences, Univ. of Honolulu, Hawaii (Jan., 1970).

109 Tucker, S.G. Emulation of Large Systems. CACM 8,12 (Dec., 1965).

110 Tucker, A.B. and Flynn, M.J. Dynamic microprogramming: processor organization and programming. CACM 14 (April, 1971), pp. 240-250.

111 Van Horn, E.C. Computer design for asynchronously reproducible multiprocessing. MAC-TR-34 (November, 1966).

112 Watson, R.W. Timesharing system design concepts. McGraw Hill, N.Y. (1970).

113 Winett, J.M. Virtual machines for developing systems software. Proc. IEEE Comput. Soc. Conf. Boston, Mass. (Sept., 1971).

114 Wirth, N. On multiprogramming, machine coding, and computer organization. CACM 12, 9 (Sept., 1969).